



LIBRARY OF THE  
UNIVERSITY OF ILLINOIS  
AT URBANA-CHAMPAIGN

510.84

IL6r

no.475-480

cop.2





Digitized by the Internet Archive  
in 2013

<http://archive.org/details/illiacciicompute475nord>









ILLIAC III COMPUTER SYSTEM MANUAL:  
TAXICRINIC PROCESSOR  
VOLUME 2

by

Bernard J. Nordmann, Jr.

August 24, 1971

THE LIBRARY OF THE

SEP 30 1971

UNIVERSITY OF ILLINOIS  
AT URBANA-CHAMPAIGN



DEPARTMENT OF COMPUTER SCIENCE  
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS



Report No. 475

ILLIAC III COMPUTER SYSTEM MANUAL:  
TAXICRINIC PROCESSOR  
VOLUME 2

by

Bernard J. Nordmann, Jr.

August 24, 1971

Department of Computer Science  
University of Illinois  
Urbana, Illinois 61801

This work was supported by Contract AT(11-1)-1018 with the U.S. Atomic Energy Commission through September 30, 1970. Current Support is under Contract AT(11-1)-2118 with the above agency.





510.84  
IL6n  
no. 475-480  
copy 2

#### ACKNOWLEDGEMENT

The conceptual design of the Taxicrinic Processor of the Illiac III Computer System is largely the work of three individuals: Roger E. Wieger, Bruce H. McCormick and the author. Dr. R. M. Lansford emphasized the importance of the associative addressing scheme wherein the processor can automatically enter the Segment Name Table for additional base descriptors, as needed. This facility, central to an effective Operating System, has been incorporated into the design of the Taxicrinic Processor.

Familiarity with the Illiac III Reference Manual, Volumes 1, 2, and 4, edited by B. H. McCormick and B. J. Nordmann, Jr., 1971, is assumed.

Bruce H. McCormick and John P. O'Donnell assisted in the editing of this volume of the TP Manual. Thanks are also due to Mrs. Roberta Andre' and Mrs. Judy Arter for their labor on the many revisions of this volume and to Stan Zundo and the countless other members of the drafting department who sweated blood over the drawings and flow charts. Finally, a special thanks goes to Dennis Reed and his magical ITEK machine for the laborious task of the reduction and printing of the flow charts without which this manual would have been impossible.



## 4. CONTROL SEQUENCES FOR THE BASIC MACHINE

### 4.1 Main Control

#### 4.1.1 Summary of Instruction Formats

##### 4.1.1.1 Primitive Instruction Formats

##### 4.1.1.2 Imprimitive Instruction Formats

##### 4.1.1.3 Operand Phrases

#### 4.1.2 Main Control Sequence

##### 4.1.2.1 Main Control Sequence Description

##### 4.1.2.2 MAIN Control Logic

#### 4.1.3 Primitive Sequence

##### 4.1.3.1 Primitive Sequence Description

##### 4.1.3.2 PRIMITIVE Control Logic

#### 4.1.4 Final Control Sequence

##### 4.1.4.1 Final Control Sequence Description

##### 4.1.4.2 FINAL Control Logic





## 4.2 Memory Access

### 4.2.1 Modes of Memory Access

- 4.2.1.1 Contiguous/Partitioned Storage Organization
- 4.2.1.2 Memory Sequence Entry Points

### 4.2.2 Constituent Tasks of Memory Access

- 4.2.2.1 Cell Alignment Check
- 4.2.2.2 Base Register Check
- 4.2.2.3 Address Bounds Check
  - 4.2.2.3.1 Address Bounds Check - Functional Description
  - 4.2.2.3.2 Address Bounds Check - Logic Description
- 4.2.2.4 Access Privilege Bits
- 4.2.2.5 Partitioned Mode Address Construction
- 4.2.2.6 Read/Write Byte Generation

### 4.2.3 Memory Access Sequence

- 4.2.3.1 Memory Access Sequence Description
- 4.2.3.2 Initial Address Construction Control Logic
- 4.2.3.3 QUEUE COUNTER UPDATE Control Logic
- 4.2.3.4 PARTITIONED MODE ADDRESS CONVERSION Control Logic
- 4.2.3.5 Memory READ Sequence Control Logic
- 4.2.3.6 Memory WRITE Sequence Control Logic
- 4.2.3.7 Memory Access Interrupt Sequence Description



### 4.3 Pointer Stack Operations

#### 4.3.1 Available Space Sequences

4.3.1.1 Available Space Sequence Descriptions

4.3.1.2 AS GET Control Logic

4.3.1.3 AS RESTORE Control Logic

#### 4.3.2 Pointer Stack Sequences

4.3.2.1 Pointer Stack Sequence Descriptions

4.3.2.2 STACK PR Control Logic

4.3.2.3 UNSTACK PR Control Logic



#### 4.4 Phrase Processing

##### 4.4.1 Phrase Process Sequence

###### 4.4.1.1 Phrase Process Sequence Description

###### 4.4.1.2 PHRASE PROCESS Control Logic

##### 4.4.2 Phrase Operation Sequence

###### 4.4.2.1 Phrase Operation Sequence Description

###### 4.4.2.2 Phrase Operation Control Logic

##### 4.4.3 Post-Operation Sequence

###### 4.4.3.1 Post-Operation Sequence Description

###### 4.4.3.2 POST-OP Control Logic

##### 4.4.4 Increment ICT Sequence

###### 4.4.4.1 Increment ICT Sequence Description

###### 4.4.4.2 INCR ICT Control Logic

##### 4.4.5 IBR Reload Sequence

###### 4.4.5.1 IBR Reload Sequence Description

###### 4.4.5.2 IBR RELOAD Control Logic

##### 4.4.6 Exchange PR-SBR Sequence

###### 4.4.6.1 Exchange PR-SBR Sequence Description

###### 4.4.6.2 Exchange PR-SBR Control Logic





## 4.5 Operand Stack Operations

### 4.5.1 Basic OS Sequences

4.5.1.1 Basic OS Sequence Descriptions

4.5.1.2 OS ENTRY Control Logic

4.5.1.3 SCR MOD Control Logic

4.5.1.4 OS READ Control Logic

4.5.1.5 OS WRITE Control Logic

### 4.5.2 Supplementary OS Sequences

4.5.2.1 Supplementary OS Sequence Descriptions

4.5.2.2 OS CLEAR Control Logic

4.5.2.3 OS INITIALIZE Control Logic



## 4.6 Interrupt Sequencing

### 4.6.1 Local Interrupts

- 4.6.1.1 Local Interrupt Design Philosophy
- 4.6.1.2 Interrupt Storage Segment
- 4.6.1.3 Interrupt Sequence
  - 4.6.1.3.1 Interrupt Sequence Description
  - 4.6.1.3.2 Interrupt Status Collection Logic
- 4.6.1.4 Increment and Check Sequence
  - 4.6.1.4.1 Increment and Check Sequence Description
- 4.6.1.5 Interrupt Return Sequence
  - 4.6.1.5.1 Interrupt Return Sequence Description
  - 4.6.1.5.2 Interrupt Status Restoration Logic
- 4.6.1.6 AS Core GET Sequence
  - 4.6.1.6.1 AS Core GET Sequence Description
- 4.6.1.7 AS Core Restore Sequence
  - 4.6.1.7.1 AS Core Restore Sequence Description
- 4.6.1.8 SLEEP Sequence
  - 4.6.1.8.1 SLEEP Sequence Description





## 4.7 Display Console and Manual Intervention

### 4.7.1 General Philosophy

### 4.7.2 Engineering Console Commands to the TP

4.7.2.1 Set Instruction Halt

4.7.2.2 Reset Instruction Halt

4.7.2.3 Set Maintenance Halt

4.7.2.4 Reset Maintenance Halt

4.7.2.5 Set Sequence Halt

4.7.2.6 Reset Sequence Halt

4.7.2.7 Execute Sequence

4.7.2.8 Interrupt

4.7.2.9 Interrupt Return

4.7.2.10 Run

4.7.2.11 Load/Read Registers

4.7.2.12 Autoload

4.7.2.13 Position

### 4.7.3 TP-Engineering Console Interface - Logical Design

4.7.3.1 Engineering Console Command Decoding

4.7.3.2 Engineering Console Command Execution



## 4.8 Turn-On and Initialization

### 4.8.1 Power Turn On

### 4.8.2 Register Clearing

### 4.8.3 TP Hardware Initialization

### 4.8.4 TP System Initialization



#### 4.A CONTROL POINT LOGICAL DESIGN

4.A.1 Design of the Control Point

4.A.2 The Use of Standard Control Points

4.A.3 The Use of Calling Control Points

4.A.4 The Design of Control Sequences Using Control Points



#### 4. Control Sequences for the Basic Machine

This section and the one following describe the control sequences used in the Taxicrimic Processors. Section 4 will be devoted to describing the sequences and logic used in the "basic machine". This includes all those sequences used to control the basic TP operations concerned with the Operand Stack, Pointer Registers, etc. It also includes the logic for handling interrupts, console processor commands, and the start-up process. However, it does not contain any of the TP instruction sequences themselves. These will be described in Section 5 which is contained in Volume III of the Taxicrimic Processor Manual.

The basic machine is essentially a distillation of all the sequences commonly used by a variety of TP instructions. These sequences operate in much the same manner as subroutines in that, when needed, they are "called" by some higher sequence. The general philosophy behind this process as well as a description of the specific logic used to implement these calls is given in Appendix 4.A at the end of this section.

Section 4 of the manual is divided into subsections on the basis of functional grouping of sequences. These groups consist of the Main Control sequences, the Memory sequences, the Pointer Stack sequences, the Phrase Processing sequences, the Operand Stack sequences, the Interrupt sequences, the Display Console control logic and the Initialization and turn-on logic.

#### 4.1 Main Control

The Main Control logic supervises the basic instruction fetch procedure. It handles the decoding of the next instruction and then chooses the proper sequence for processing it.



#### 4.1.1 Summary of Instruction Formats

Every Illiac III instruction can be considered to be in prefix form: an operation (specified by a mnemonic byte) followed by operands (designated by operand phrases), if any. An instruction with  $n$  operands in main storage has  $n$  operand phrases respectively, each one implicitly specifying the data address of the operand. In addition, an instruction may call upon operands from the Operand Stack. In this case, the operand address is implied by the mnemonic byte.

The general format for the Illiac III instructions is shown in figure 4.1.1.

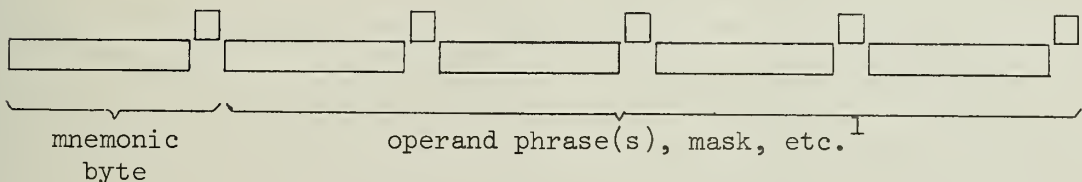


Figure 4.1.1 Illiac III Instruction Format

This format consists of a single mnemonic byte normally followed by one or more operand phrases. These phrases may be long or short (see Section 2.1.2).

In primitive instructions the total length of these operand phrases may not exceed 4 bytes. Some few primitive instructions, while adhering to the 4-byte constraint, have fields with alternate interpretation: mask, etc.

In imprimitive instructions there may be up to 12 operand phrases. Here however, no restriction is placed upon the number of phrases which may be long.

Operand phrases provide a uniform technique throughout Illiac III for addressing main storage and for operating on the 15

---

<sup>1</sup> For imprimitive instructions up to 12 operand phrases are allowed (36 bytes max.).

pointer stacks. In an operand phrase, the file (main store) address of an operand is implied by giving the name of its associated pointer stack. That is, the data address is specified by the topmost pointer in the pointer stack named by the operand phrase tag field.

In addition to naming a pointer stack, an operand phrase may also specify operations which modify the value of the pointer and/or change the depth of the pointer stack. These operations take place before, after, or both before and after the actual execution of the instruction.

#### 4.1.1.1 Primitive Instruction Format

Primitives correspond to the "machine language" of most other computer systems and are by far the most prevalent instruction-type.

Primitive instructions always begin with a single-byte phrase, the mnemonic phrase, which serves to name the instruction and to identify the number and format of any following associated phrase(s) since they are employed to construct the address(es) of operand cell(s). Other terminating phrases, such as counts or masks are also possible and are discussed in detail in later sections.

#### 4.1.1.2 Imprimitive Instruction Formats

These are machine-oriented instructions, reducible without programmed intervention to a nested sequence of imprimitive and primitive instructions, and ultimately to a sequence of primitive instructions. Imprimitives operate only on the pointer registers: renaming, storing, and modifying them. This class of instructions is also used to accomplish transfers of control, both conditional and unconditional.

Imprimitive instructions always begin with a mnemonic byte. As is appropriate for the particular instruction, none or more operand phrases may follow the mnemonic byte.

The length of each operand phrase (1 or 3 bytes) is specified by the flag of the first byte of the phrase: a flag of '0' indicates a short phrase; a flag of '1' indicates a long phrase. The terminal phrase is specified by the low order bit of this same byte: '0' if more phrases follow, '1' for the terminal phrase. Accordingly, the instruction field is consecutively partitioned into phrases until a terminal bit is sensed (imprimitives).

### 4.1.1.3 Operand Phrases

Every operand, other than those in the top of the Operand Stack, is designated by use of an Operand Phrase. The length of each operand phrase (1 or 3 bytes) is specified by the flag of the first byte in each phrase: a flag of '0' indicates a short phrase; a flag of '1' indicates a long phrase.

The fields of the two types of operand phrases are shown in Figure 4.1.1.3.

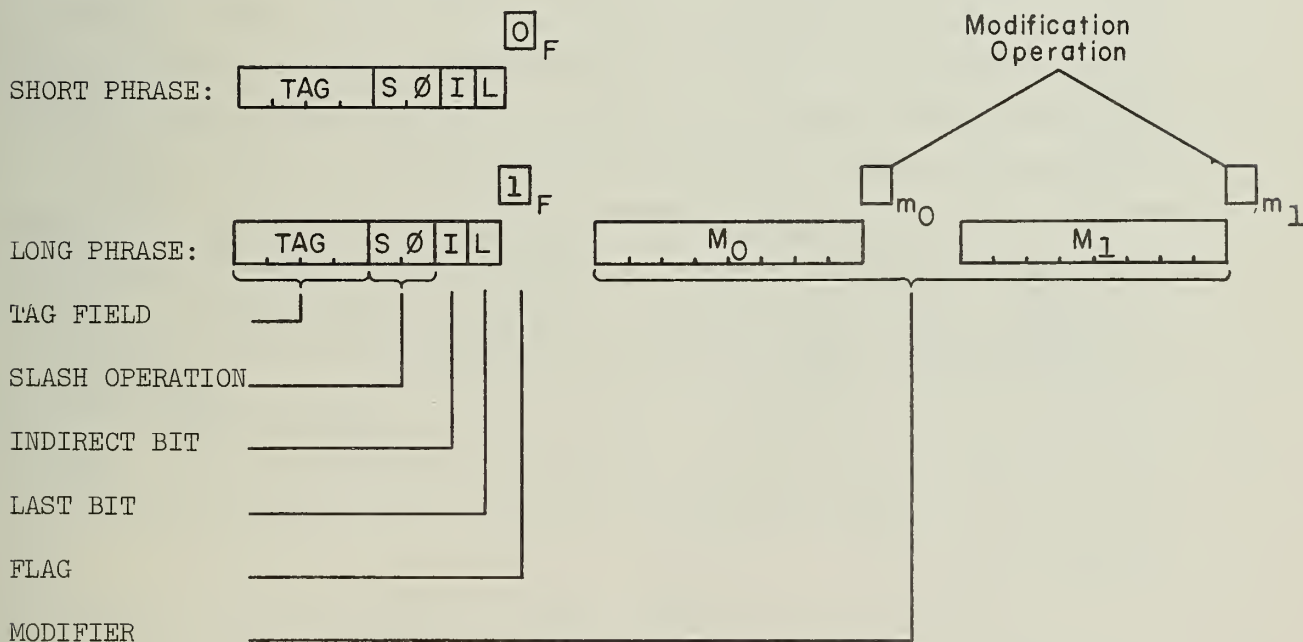


Figure 4.1.1.3 Operand Phrase Format

#### 4.1.2 Main Control Sequence

##### 4.1.2.1 Main Control Sequence Description

The Main Control Sequence supervises all of the other sequences in the machine. It loads the mnemonic register with the instruction code, decodes the instruction, and establishes the instruction sequence (Section 5) which must be performed. In addition, the Main Control Sequence detects interrupts, executes and any other exceptional activities which may occur.

The Main Control Sequence factors into 2 distinct parts: the Primitive Sequence and the Imprimitive Sequence, both terminated by a common Final Control Sequence (see Figure 4.1.2.1/1). Depending on the type of instruction, one of these first two sequences is called to decode and determine the execution of the instruction under consideration. The overwhelming majority of the instructions executed by the TP are Primitive Instructions, whose sequences are described in Sections 5.1-5.4. The Imprimitive Instruction Sequence is described in Section 5.5.

The first task of the Main Control Sequence when a new instruction is to be executed is to load the IR from the Instruction Buffer Register. Since every instruction ends by making sure that the IBR contains at least one byte of the next instruction, the IR is guaranteed to contain at least the mnemonic byte. The IR is loaded so that the instruction's mnemonic byte is in the rightmost byte position and the remaining bytes are in sequence beginning at the left. This format is shown in Figure 4.1.2.1/2.

After the IR has been loaded, the mnemonic byte is gated to the mnemonic byte register where it is stored for the rest of the instruction execution sequence. The mnemonic byte register is used to drive most of the control signals for the instruction sequences. The output of the mnemonic register is automatically decoded to give the various instruction type codes (zero operand, one operand, imprimitive, PAU, Arithmetic, etc.) as well as the actual activating signal for the instructions themselves. The decoding network decodes the flag and first 6 bits of the mnemonic byte into 128



output lines. These are then combined with the last two bits, if necessary, to produce the instruction activation signals. This method was chosen since many of the instructions in fact use the last two bits as number type or cell size designators and therefore, these bits do not always have to be decoded to determine the instruction. Figure 4.1.3.2/1 gives the operation codes for all of the TP instructions organized according to type.

The next step is to check the instruction for legality. If the op-code does not represent a legal instruction, an illegal instruction interrupt is performed. If a privileged instruction is attempted while the TP is in slave mode, a privilege violation interrupt is executed. The high order two bits and the flag of the mnemonic are also decoded and either the Primitive or Imprimitive Sequence is started. Observe that these three bits identify the unit which will ultimately execute the instruction: TP, AU, PAU, etc. The beginning of the Main Control Sequence is shown in the flowchart at the end of this section. The rest of this section of the manual is devoted to describing the constituent parts of the Main Control Sequence. Section 4.1.3 starts by describing the Primitive Sequence. Section 4.1.4 describes the Final Control Sequence used for the termination of both primitive and imprimitive instructions.

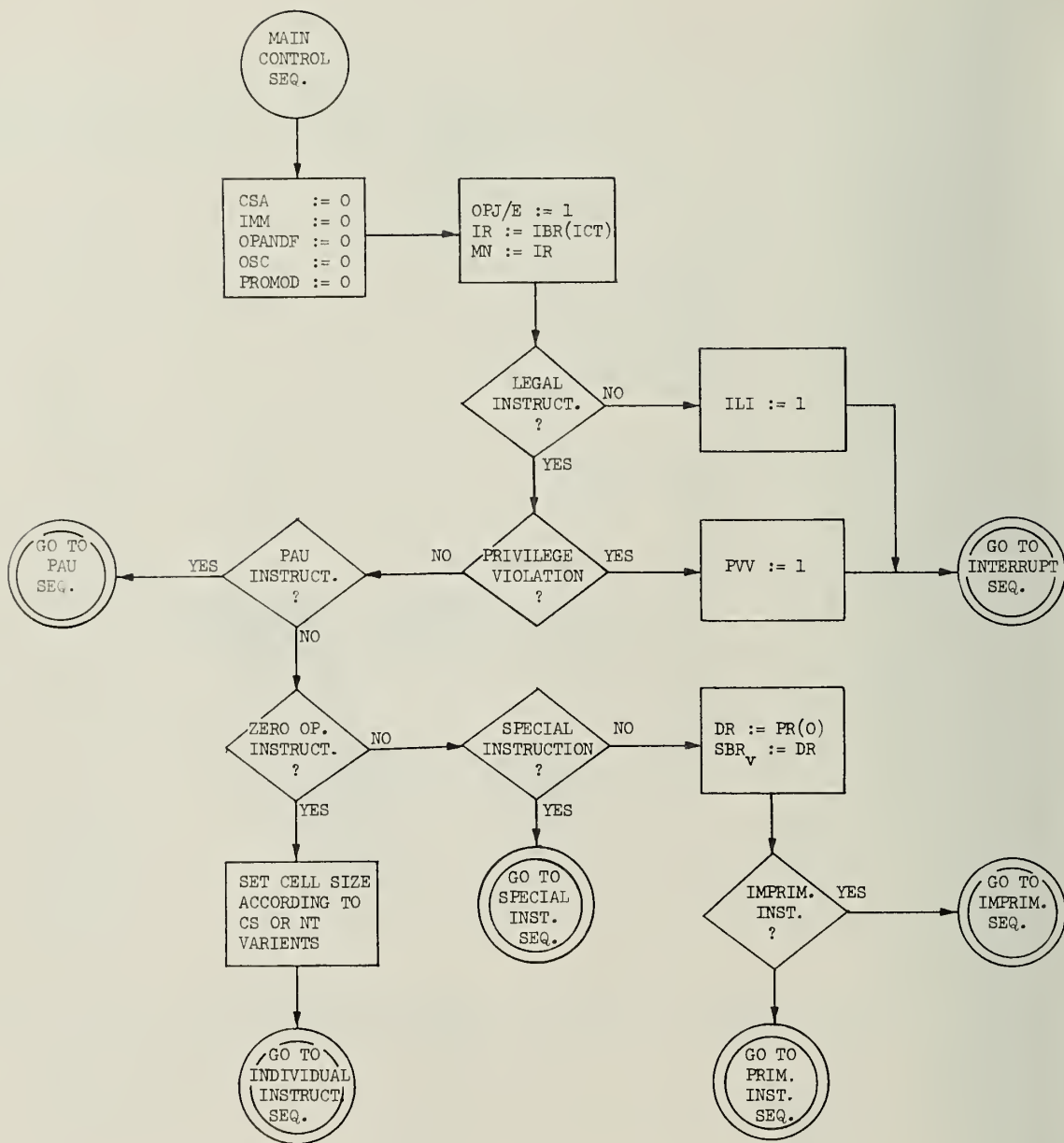




O p e r a n d	P h r a s e	M n e m o n i c
---------------	-------------	-----------------

Primitive Instruction Left Justified  
on Byte 1 of Instruction

Figure 4.1.2.1/2 - Format for Instructions in IR

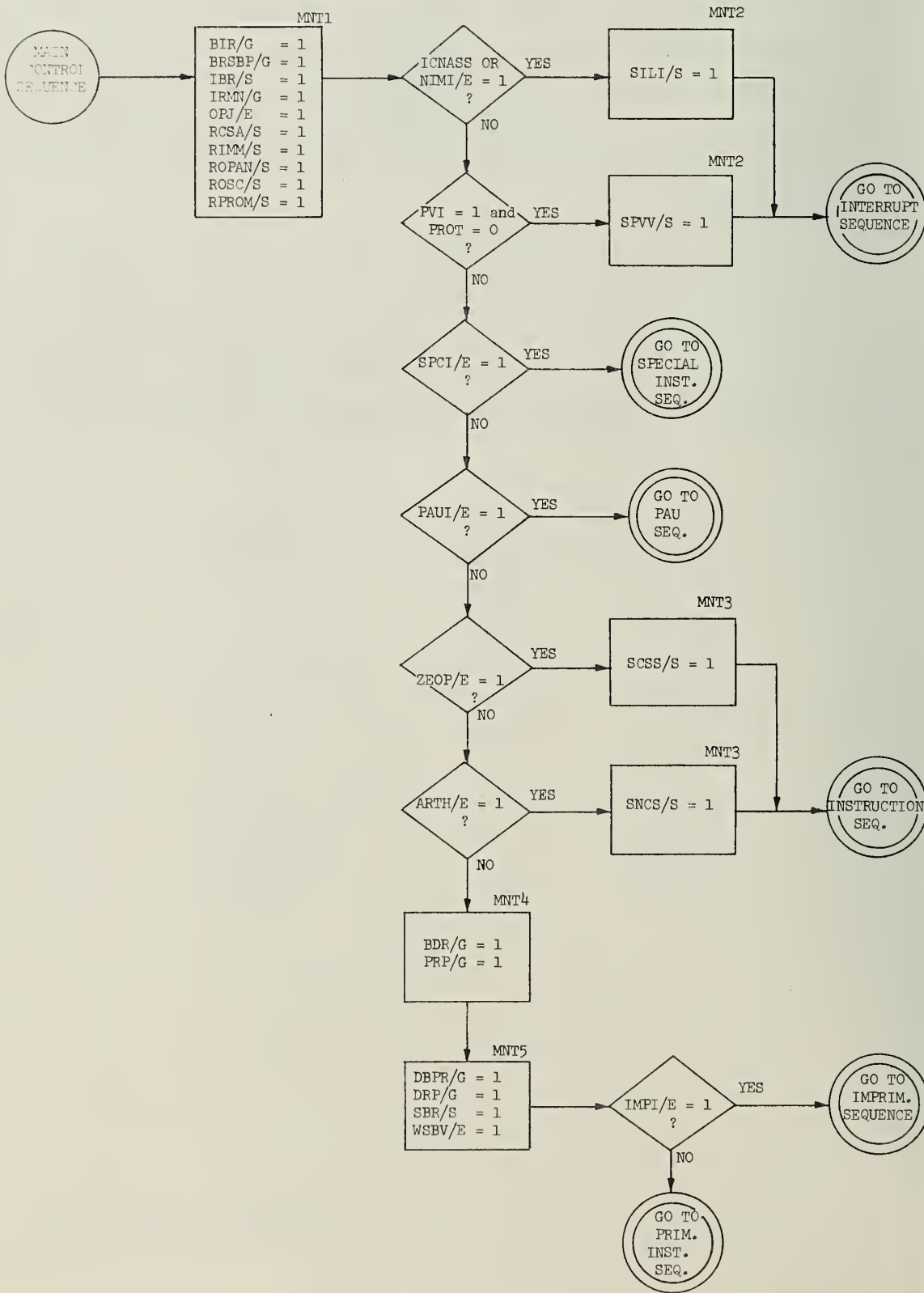


MAIN Control Sequence

#### 4.1.2.2 Main Control Logic

Control point MNT1 is used to load the IR from the IBR and to load the mnemonic byte register from the IR. Then a check is made for an illegal opcode or the attempted use of a privileged instruction. In the former case, the indicator SILI/S is turned on, while in the later, SPVV/S is activated before control is given to the interrupt sequence.

The signals SPCI/E, PAUI/E, ZEOP/E, ARTH/E and IMPI/E steer control to one of five sequences for further processing. In designing the logic necessary to make decisions regarding the above signals, it was tacitly assumed that they were mutually exclusive events (ZEOP/E and ARTH/E were combined into one signal).



Main Control Sequence

#### 4.1.3 Primitive Sequence

The Primitive Sequence is the initial sequence used to process all primitive instructions. The basic purposes of this sequence are:

- 1) to ensure that the IR is loaded with the operand phrases before the instruction is executed,
- 2) to perform the required phrase operations as they are needed,
- 3) to place the initial byte of each phrase in a standard format in the IR after the phrase operations have been performed, and modifiers are no longer needed,
- 4) to take care of various contingencies such as modification of PR#13 or conditional subtraction failure before instruction execution begins.

The following sections give a detailed description of the operation and the control logic of the Primitive Sequence. In Section 4.1.3.1 the operation will be described in two distinct phases in order to separate out the effects of interrupts. The reason for this is that in this sequence the interrupt returns become rather confusing.

#### 4.1.3.1 Primitive Sequence Description

Once the Main Control Sequence has determined that the instruction to be executed is a Primitive Instruction, the Primitive Sequence's first task is to determine whether or not the complete instruction is in the IR. This can be done by knowing where the ICT is "pointing" in the IBR and how long the instruction is. The length is determined by the instruction type (i.e. how many operands) and whether the operands are long or short.

If the IR must be refilled, the ICT is first incremented by one. This enables the IR to be loaded without the mnemonic byte. In the case of a two operand instruction with one long phrase, this is extremely important since otherwise the IR cannot contain all of the phrase data. After the ICT is incremented it is checked once again to see if the IBR really needs to be reloaded or if it is only necessary to reload the IR from the IBR. When the IR (and possibly the IBR) has been reloaded the ICT must be restored to its original value so that any operations involving PR#0 are given the correct PR#0 value. This is done by incrementing the ICT by 7 and resetting the ICT overflow bit (there is no subtraction on the ICT).

Next, the phrase processing operations are performed. For the one operand phrase instruction this is very straightforward. The first operand phrase is processed by the Phrase Process Sequence as soon as the IBR and IR are loaded and the phrase is then finished.

For the three operand phrase instruction, the sequence is fairly reasonable since each phrase must be short. After the first phrase has been processed, the IR is permuted left by one byte and the second phrase is processed by repeating the phrase sequence. The third phrase is handled in the same manner, after which the IR is returned to the format shown in Figure 4.1.3.1/1

The two operand phrase primitive instructions present a slightly more difficult problem. After the first phrase has been processed with the phrase process sequence, the first two bytes after the mnemonic are checked to see if the first phrase of the two phrase instruction is short. If it is, the same sequence that is used for the



three operand phrase instructions can be used, except that the phrase processing sequence for the third phrase is skipped and the IR is rotated one extra byte after the second phrase has been processed. At the end of this sequence the IR will have the format shown in Figure 4.1.3.1/2.

If the first of the two operands in the instruction is long, we must perform a certain amount of shifting in the IR before the second phrase can be processed. Note the format of the IR at this point (see Figure 4.1.3.1/3). In order to arrange the IR so that the second phrase can be processed, the third byte of the IR (the leftmost byte, 2a) is gated into the first byte position (i.e. where 1b originally was located). Note that this destroys the modifier portion of the first phrase. However, since this has already been processed, it does not matter anymore. The IR format now conforms to Figure 4.1.3.1/2 except that the leftmost 2 bytes contain garbage.

Once the IR is in this format it can be processed using the same procedure that is used for two operand phrase instructions with the first phrase short.

If PR#13 has been modified by the phrase sequence, the OS will have been cleared out and OSC set to 1. Before the instruction execution can begin the OS must therefore be initialized.

If there was a conditional subtraction failure, CSF is set to 1 and PR#0 is reloaded with its original value which was stored in the value portion of the SBR by the Main Control Sequence. Otherwise, CSA is checked to see if any conditional subtractions were attempted. If CSA = 1, a successful subtraction occurred and the CSF indicator is set to 0. If no conditional subtraction was attempted CSF is left alone.

Finally, if the instruction has an immediate address option and it is specified, the field designator bits must be interpreted in terms of PL, PV, PR or SN instead of cell size. Then control is given to the sequence specified by the decoded mnemonic byte in the IR.

1st phrase	2nd phrase	3rd phrase	-
byte 0	byte 1	byte 2	byte 3

Figure 4.1.3.1/1 Format of IR at End of  
Primitive Sequence with Three Short Phrases

1st phrase	2nd phrase	2nd phrase	2nd phrase
byte 0	byte 1	byte 2	byte 3

Figure 4.1.3.1/2 Format of IR at End of  
Primitive Sequence if Second Phrase is Long

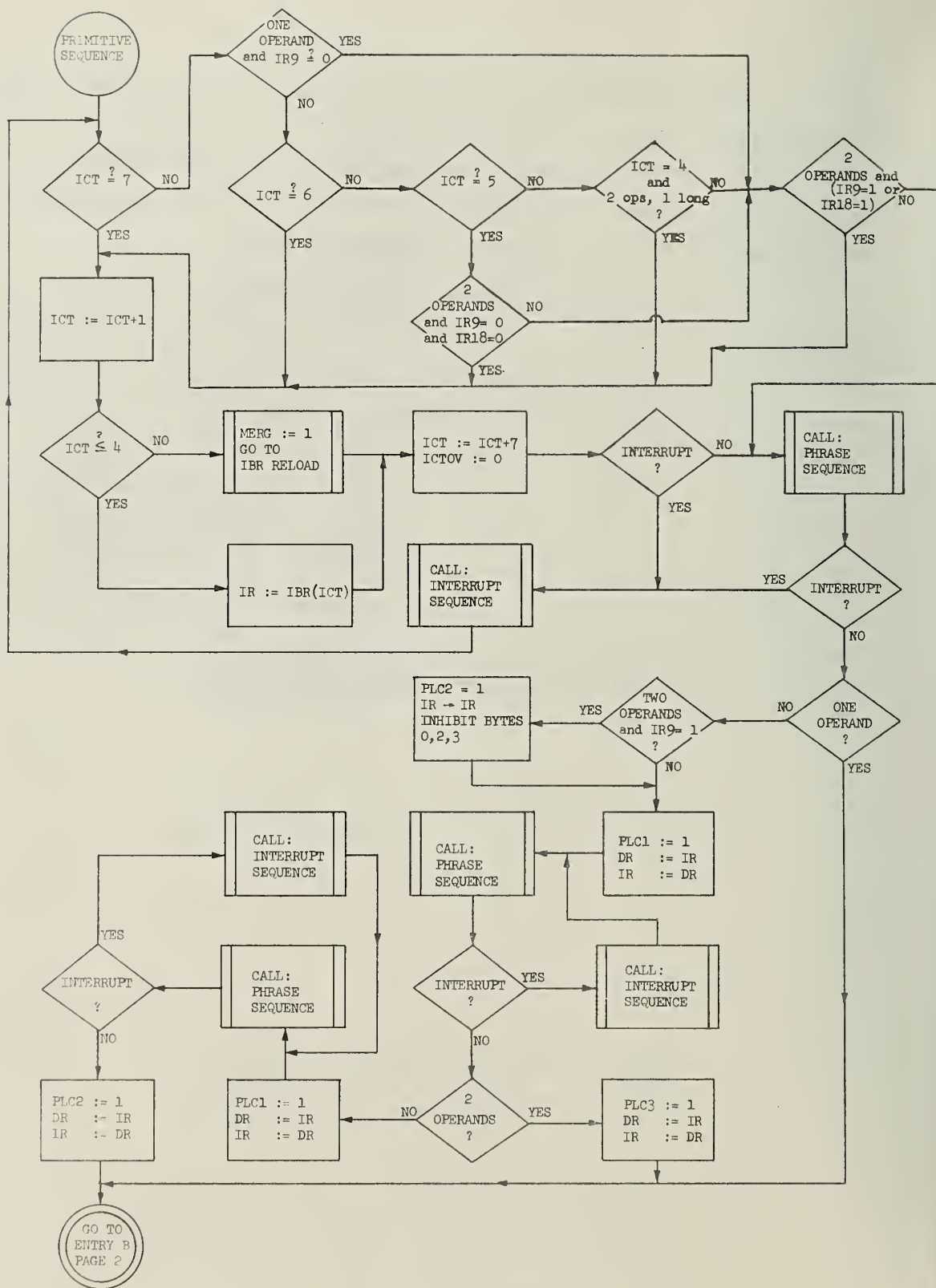
1st phrase	1st phrase	1st phrase	2nd phrase
byte 0	byte 1	byte 2	byte 3

Figure 4.1.3.1/3 Format of IR After the First  
Phrase is Processed if the First Phrase is Long

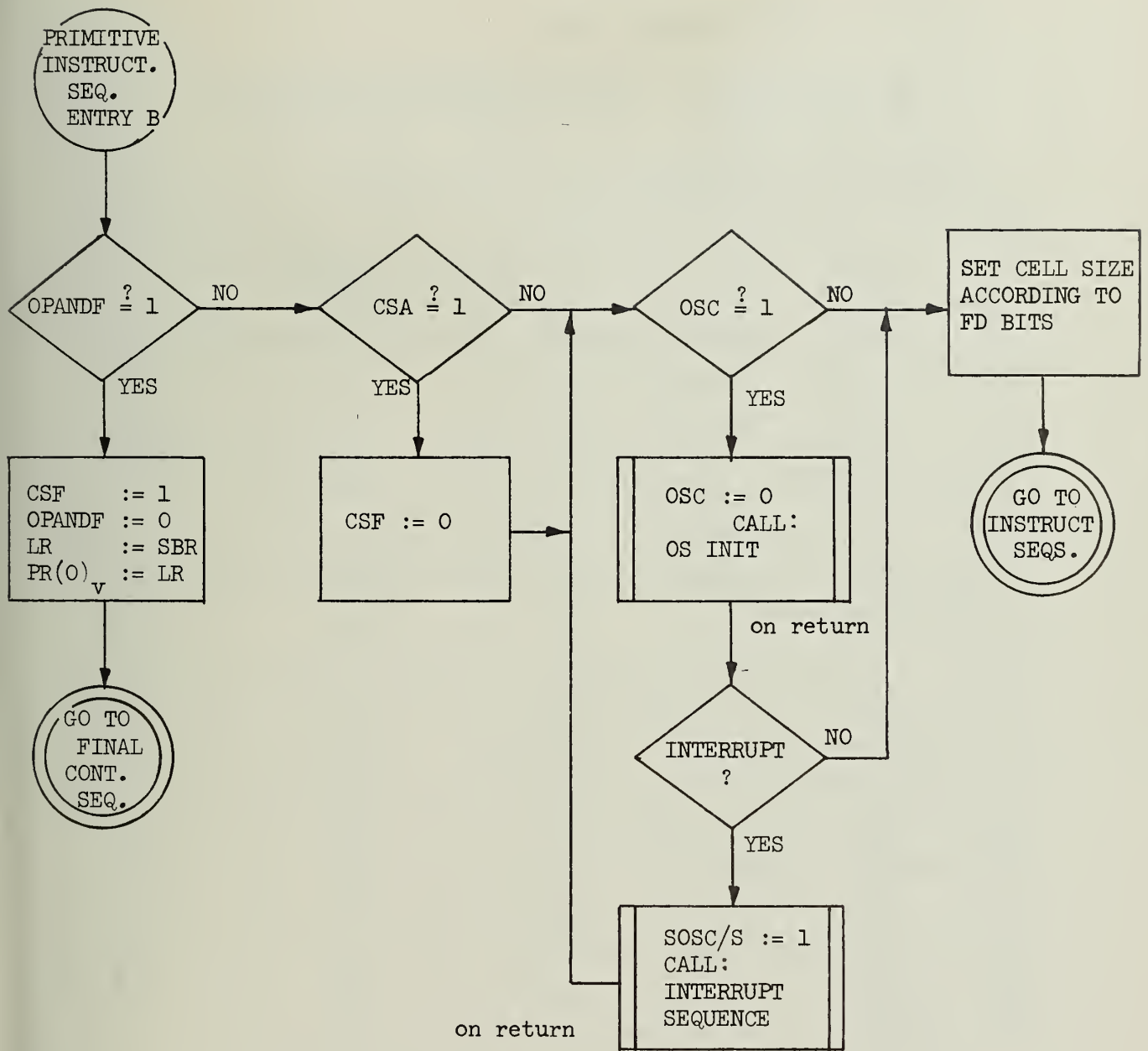


The treatment of interrupts in this sequence becomes fairly complicated. This is due to the fact that because of the many phrase modifications, the sequence is not restartable from the beginning if an interrupt occurs in the middle of the sequence. In fact, there are three possible interrupt return restarting points.

If an interrupt occurs in either the IBR Reload or the first Phrase Process Sequence, the complete instruction can be restarted. If an interrupt occurs in second Phrase Process Sequence, the interrupt must return to the point immediately preceeding it and attempt to perform it again when the situation has been fixed. If an interrupt occurs in the third Phrase Sequence, the interrupt must eventually return and repeat that sequence. Finally, if an interrupt occurs during the OS Initialize Sequence, the interrupt will return to the point just before the decision to initialize the OS. This point is also used as the normal interrupt return point for most interrupts which may occur during the execution of the Primitive Instructions.



Primitive Sequence



Primitive Sequence Flow Chart

#### 4.1.3.2 PRIMITIVE Control Logic

The control logic for the Primitive Instruction Control Sequence is fairly complicated. As can be seen in the flowchart at the end of this section there are several signals which must be decoded directly from the mnemonic byte itself. These include.

- 1) No operand phrase instruction, ZEOP/E
- 2) One operand phrase instruction, ONOP/E
- 3) Two operand phrase instruction, TWOP/E
- 4) "Complex" instruction, CXOP/E

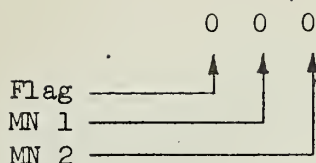
Fig. 4.1.3.2/1 shows a listing of all mnemonic phrases used and how they are categorized according to this scheme.

The signals which indicate the state of the ICT are easily derived from the ICT bit signals using the standard decoder shown in Figure 4.1.3.2/2.

The sequence begins by activating control point PIT1. This control point is used as a "null" control point to generate the initial starting signal of the sequence. Note that if it were not present there would be an initial decision, based on the state of RLBR, as to which control point to activate: PIT2, PIT3, or PIT9. However, if PIMSTRT, the Primitive Sequence start signal, were used to activate this decision and PIT2 or PIT3 were chosen as a result, the ICT1/E signal would be activated. This signal causes an incrementation of the ICT which in turn might cause a change in state in RLBR. If PINSTRT is still on when this change occurs (and this is true in the general case) then PIT9 will also be turned on much earlier than it would have any right to be.

In order to prevent this, PIT1 is used to generate an advance out signal which will stay active long enough to select the proper control point but which will also turn off way before RLBR changes state.

# ZERO OPERAND INSTRUCTION CODES -



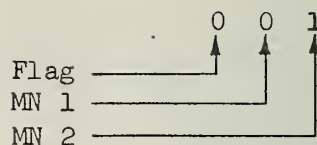
MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION	MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION
0 0 0 0 0 0	ZERO (B)	1 0 0 0 0 0	SLUFF (B)
0 0 0 0 0 1	ZERO (H)	1 0 0 0 0 1	SLUFF (H)
0 0 0 0 1 0	ZERO (W)	1 0 0 0 1 0	SLUFF (W)
0 0 0 0 1 1	ZERO (D)	1 0 0 0 1 1	SLUFF (D)
0 0 0 1 0 0	ONE (B)	1 0 0 1 0 0	DUP (B)
0 0 0 1 0 1	ONE (H)	1 0 0 1 0 1	DUP (H)
0 0 0 1 1 0	ONE (W)	1 0 0 1 1 0	DUP (W)
0 0 0 1 1 1	ONE (D)	1 0 0 1 1 1	DUP (D)
0 0 1 0 0 0	COUNT (B)	1 0 1 0 0 0	XCH (B)
0 0 1 0 0 1	COUNT (H)	1 0 1 0 0 1	XCH (H)
0 0 1 0 1 0	COUNT (W)	1 0 1 0 1 0	XCH (W)
0 0 1 0 1 1	COUNT (D)	1 0 1 0 1 1	XCH (D)
0 0 1 1 0 0	BIT (B)	1 0 1 1 0 0	CPRL (B)
0 0 1 1 0 1	BIT (H)	1 0 1 1 0 1	CPRL (H)
0 0 1 1 1 0	BIT (W)	1 0 1 1 1 0	CPRL (W)
0 0 1 1 1 1	BIT (D)	1 0 1 1 1 1	CPRL (D)
0 1 0 0 0 0	AND (B)	1 1 0 0 0 0	NOT (B)
0 1 0 0 0 1	AND (H)	1 1 0 0 0 1	NOT (H)
0 1 0 0 1 0	AND (W)	1 1 0 0 1 0	NOT (W)
0 1 0 0 1 1	AND (D)	1 1 0 0 1 1	NOT (D)
0 1 0 1 0 0	OR (B)	1 1 0 1 0 0	ACTP (#0)
0 1 0 1 0 1	OR (H)	1 1 0 1 0 1	ACTP (#1)
0 1 0 1 1 0	OR (W)	1 1 0 1 1 0	ACTP (#2)
0 1 0 1 1 1	OR (D)	1 1 0 1 1 1	ACTP (#3)
0 1 1 0 0 0	XOR (B)	1 1 1 0 0 0	INRT
0 1 1 0 0 1	XOR (H)	1 1 1 0 0 1	
0 1 1 0 1 0	XOR (W)	1 1 1 0 1 0	LTR
0 1 1 0 1 1	XOR (D)	1 1 1 0 1 1	STR
0 1 1 1 0 0	EQV (B)	1 1 1 1 0 0	SVC
0 1 1 1 0 1	EQV (H)	1 1 1 1 0 1	SVR
0 1 1 1 1 0	EQV (W)	1 1 1 1 1 0	
0 1 1 1 1 1	EQV (D)	1 1 1 1 1 1	WHO

Figure 4.1.3.2/1 Instruction Mnemonic Codes

6/11/71

Section 4.1.3.2 - 2/12

ONE OPERAND INSTRUCTION CODES -

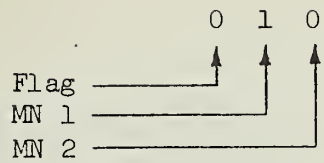


MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION	MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION
0 0 0 0 0 0	PUSH (B)	1 0 0 0 0 0	LS (B)
0 0 0 0 0 1	PUSH (H)	1 0 0 0 0 1	LS (H)
0 0 0 0 1 0	PUSH (W)	1 0 0 0 1 0	LS (W)
0 0 0 0 1 1	PUSH (D)	1 0 0 0 1 1	
0 0 0 1 0 0	LD (B)	1 0 0 1 0 0	RS (B)
0 0 0 1 0 1	LD (H)	1 0 0 1 0 1	RS (H)
0 0 0 1 1 0	LD (W)	1 0 0 1 1 0	RS (W)
0 0 0 1 1 1	LD (D)	1 0 0 1 1 1	
0 0 1 0 0 0	POP (B)	1 0 1 0 0 0	
0 0 1 0 0 1	POP (H)	1 0 1 0 0 1	
0 0 1 0 1 0	POP (W)	1 0 1 0 1 0	
0 0 1 0 1 1	POP (D)	1 0 1 0 1 1	
0 0 1 1 0 0	ST (B)	1 0 1 1 0 0	
0 0 1 1 0 1	ST (H)	1 0 1 1 0 1	
0 0 1 1 1 0	ST (W)	1 0 1 1 1 0	
0 0 1 1 1 1	ST (D)	1 0 1 1 1 1	
0 1 0 0 0 0	SET (B)	1 1 0 0 0 0	
0 1 0 0 0 1	SET (H)	1 1 0 0 0 1	
0 1 0 0 1 0	SET (W)	1 1 0 0 1 0	
0 1 0 0 1 1	SET (D)	1 1 0 0 1 1	
0 1 0 1 0 0	RESET (B)	1 1 0 1 0 0	
0 1 0 1 0 1	RESET (H)	1 1 0 1 0 1	
0 1 0 1 1 0	RESET (W)	1 1 0 1 1 0	
0 1 0 1 1 1	RESET (D)	1 1 0 1 1 1	
0 1 1 0 0 0	TEST (B)	1 1 1 0 0 0	
0 1 1 0 0 1	TEST (H)	1 1 1 0 0 1	
0 1 1 0 1 0	TEST (W)	1 1 1 0 1 0	
0 1 1 0 1 1	TEST (D)	1 1 1 0 1 1	
0 1 1 1 0 0	TESTM (B)	1 1 1 1 0 0	SLEEP
0 1 1 1 0 1	TESTM (H)	1 1 1 1 0 1	INCK
0 1 1 1 1 0	TESTM (W)	1 1 1 1 1 0	SIM
0 1 1 1 1 1	TESTM (D)	1 1 1 1 1 1	

Figure 4.1.3.2/1 continued Instruction Mnemonic Codes



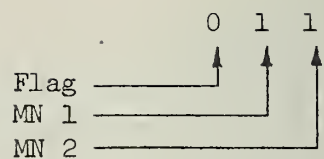
PRESENTLY UNUSED INSTRUCTION CODES -



MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION	MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION
0 0 0 0 0 0		1 0 0 0 0 0	
0 0 0 0 0 1		1 0 0 0 0 1	
0 0 0 0 1 0		1 0 0 0 1 0	
0 0 0 0 1 1		1 0 0 0 1 1	
0 0 0 1 0 0		1 0 0 1 0 0	
0 0 0 1 0 1		1 0 0 1 0 1	
0 0 0 1 1 0		1 0 0 1 1 0	
0 0 0 1 1 1		1 0 0 1 1 1	
0 0 1 0 0 0		1 0 1 0 0 0	
0 0 1 0 0 1		1 0 1 0 0 1	
0 0 1 0 1 0		1 0 1 0 1 0	
0 0 1 0 1 1		1 0 1 0 1 1	
0 0 1 1 0 0		1 0 1 1 0 0	
0 0 1 1 0 1		1 0 1 1 0 1	
0 0 1 1 1 0		1 0 1 1 1 0	
0 0 1 1 1 1		1 0 1 1 1 1	
0 1 0 0 0 0		1 1 0 0 0 0	
0 1 0 0 0 1		1 1 0 0 0 1	
0 1 0 0 1 0		1 1 0 0 1 0	
0 1 0 0 1 1		1 1 0 0 1 1	
0 1 0 1 0 0		1 1 0 1 0 0	
0 1 0 1 0 1		1 1 0 1 0 1	
0 1 0 1 1 0		1 1 0 1 1 0	
0 1 0 1 1 1		1 1 0 1 1 1	
0 1 1 0 0 0		1 1 1 0 0 0	
0 1 1 0 0 1		1 1 1 0 0 1	
0 1 1 0 1 0		1 1 1 0 1 0	
0 1 1 0 1 1		1 1 1 0 1 1	
0 1 1 1 0 0		1 1 1 1 0 0	
0 1 1 1 0 1		1 1 1 1 0 1	
0 1 1 1 1 0		1 1 1 1 1 0	
0 1 1 1 1 1		1 1 1 1 1 1	

Figure 4.1.3.2/1 continued Instruction Mnemonic Codes

PRESENTLY UNUSED INSTRUCTION CODES -

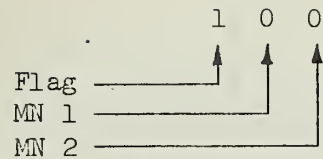


MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION	MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION
0 0 0 0 0 0		1 0 0 0 0 0	
0 0 0 0 0 1		1 0 0 0 0 1	
0 0 0 0 1 0		1 0 0 0 1 0	
0 0 0 0 1 1		1 0 0 0 1 1	
0 0 0 1 0 0		1 0 0 1 0 0	
0 0 0 1 0 1		1 0 0 1 0 1	
0 0 0 1 1 0		1 0 0 1 1 0	
0 0 0 1 1 1		1 0 0 1 1 1	
0 0 1 0 0 0		1 0 1 0 0 0	
0 0 1 0 0 1		1 0 1 0 0 1	
0 0 1 0 1 0		1 0 1 0 1 0	
0 0 1 0 1 1		1 0 1 0 1 1	
0 0 1 1 0 0		1 0 1 1 0 0	
0 0 1 1 0 1		1 0 1 1 0 1	
0 0 1 1 1 0		1 0 1 1 1 0	
0 0 1 1 1 1		1 0 1 1 1 1	
0 1 0 0 0 0		1 1 0 0 0 0	
0 1 0 0 0 1		1 1 0 0 0 1	
0 1 0 0 1 0		1 1 0 0 1 0	
0 1 0 0 1 1		1 1 0 0 1 1	
0 1 0 1 0 0		1 1 0 1 0 0	
0 1 0 1 0 1		1 1 0 1 0 1	
0 1 0 1 1 0		1 1 0 1 1 0	
0 1 0 1 1 1		1 1 0 1 1 1	
0 1 1 0 0 0		1 1 1 0 0 0	
0 1 1 0 0 1		1 1 1 0 0 1	
0 1 1 0 1 0		1 1 1 0 1 0	
0 1 1 0 1 1		1 1 1 0 1 1	
0 1 1 1 0 0		1 1 1 1 0 0	
0 1 1 1 0 1		1 1 1 1 0 1	
0 1 1 1 1 0		1 1 1 1 1 0	
0 1 1 1 1 1		1 1 1 1 1 1	

Figure 4.1.3.2/1 continued Instruction Mnemonic Codes



MISC. INSTRUCTION CODES -

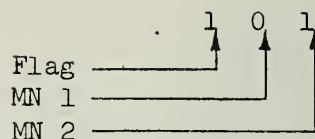


Two operands

MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION	MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION
0 0 0 0 0 0	ASSIGN	1 0 0 0 0 0	PUSHF (B)
0 0 0 0 0 1	ASSIGN	1 0 0 0 0 1	PUSHF (H)
0 0 0 0 1 0	ASSIGN	1 0 0 0 1 0	PUSHF (W)
0 0 0 0 1 1	ASSIGN	1 0 0 0 1 1	PUSHF (D)
0 0 0 1 0 0	POPF (B)	1 0 0 1 0 0	PUSHFR (B)
0 0 0 1 0 1	POPF (H)	1 0 0 1 0 1	PUSHFR (H)
0 0 0 1 1 0	POPF (W)	1 0 0 1 1 0	PUSHFR (W)
0 0 0 1 1 1	POPF (D)	1 0 0 1 1 1	PUSHFR (D)
0 0 1 0 0 0	POPFR (B)	1 0 1 0 0 0	SCAN (B)
0 0 1 0 0 1	POPFR (H)	1 0 1 0 0 1	SCAN (H)
0 0 1 0 1 0	POPFR (W)	1 0 1 0 1 0	SCAN (W)
0 0 1 0 1 1	POPFR (D)	1 0 1 0 1 1	SCAN (D)
0 0 1 1 0 0	PACK	1 0 1 1 0 0	SCANM (B)
0 0 1 1 0 1	RNAM	1 0 1 1 0 1	SCANM (H)
0 0 1 1 1 0	UNPACK	1 0 1 1 1 0	SCANM (W)
0 0 1 1 1 1	LINK	1 0 1 1 1 1	SCANM (D)
0 1 0 0 0 0	CALL	1 1 0 0 0 0	MOVE
0 1 0 0 0 1	EXECUTE	1 1 0 0 0 1	TRANS
0 1 0 0 1 0	GOTO	1 1 0 0 1 0	EDIT
0 1 0 0 1 1	EXIT	1 1 0 0 1 1	
0 1 0 1 0 0	NOP	1 1 0 1 0 0	IF
0 1 0 1 0 1	SPECIFY	1 1 0 1 0 1	IFN
0 1 0 1 1 0	LOC	1 1 0 1 1 0	
0 1 0 1 1 1		1 1 0 1 1 1	
0 1 1 0 0 0	RESU	1 1 1 0 0 0	
0 1 1 0 0 1		1 1 1 0 0 1	
0 1 1 0 1 0		1 1 1 0 1 0	
0 1 1 0 1 1		1 1 1 0 1 1	
0 1 1 1 0 0		1 1 1 1 0 0	
0 1 1 1 0 1		1 1 1 1 0 1	
0 1 1 1 1 0		1 1 1 1 1 0	
0 1 1 1 1 1		1 1 1 1 1 1	

Figure 4.1.3.2/1 continued Instruction Mnemonic Codes

ZERO AND ONE OPERAND INSTRUCTION CODES -

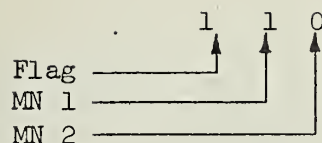


MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION	MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION
0 0 0 0 0 0	LIBR	1 0 0 0 0 0	RDCLK
0 0 0 0 0 1	SL	1 0 0 0 0 1	
0 0 0 0 1 0	SIO	1 0 0 0 1 0	STTIM
0 0 0 0 1 1	HIO	1 0 0 0 1 1	RDTIM
0 0 0 1 0 0		1 0 0 1 0 0	
0 0 0 1 0 1	SR	1 0 0 1 0 1	
0 0 0 1 1 0		1 0 0 1 1 0	
0 0 0 1 1 1		1 0 0 1 1 1	
0 0 1 0 0 0		1 0 1 0 0 0	
0 0 1 0 0 1	GET	1 0 1 0 0 1	
0 0 1 0 1 0		1 0 1 0 1 0	
0 0 1 0 1 1		1 0 1 0 1 1	
0 0 1 1 0 0		1 0 1 1 0 0	
0 0 1 1 0 1	PUT	1 0 1 1 0 1	
0 0 1 1 1 0		1 0 1 1 1 0	
0 0 1 1 1 1		1 0 1 1 1 1	
0 1 0 0 0 0		1 1 0 0 0 0	
0 1 0 0 0 1	INCL	1 1 0 0 0 1	
0 1 0 0 1 0		1 1 0 0 1 0	
0 1 0 0 1 1		1 1 0 0 1 1	
0 1 0 1 0 0		1 1 0 1 0 0	
0 1 0 1 0 1	INCR	1 1 0 1 0 1	
0 1 0 1 1 0		1 1 0 1 1 0	
0 1 0 1 1 1		1 1 0 1 1 1	
0 1 1 0 0 0		1 1 1 0 0 0	
0 1 1 0 0 1	DECL	1 1 1 0 0 1	
0 1 1 0 1 0		1 1 1 0 1 0	
0 1 1 0 1 1		1 1 1 0 1 1	
0 1 1 1 0 0		1 1 1 1 0 0	
0 1 1 1 0 1	DECR	1 1 1 1 0 1	
0 1 1 1 1 0		1 1 1 1 1 0	
0 1 1 1 1 1		1 1 1 1 1 1	

← one operand → ← zero operand →

Figure 4.1.3.2/1 continued Instruction Mnemonic Codes

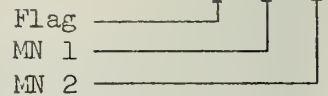
# ARITHMETIC OPERATION CODES -



MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION	MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION
0 0 0 0 0 0		1 0 0 0 0 0	ADD (SFX)
0 0 0 0 0 1		1 0 0 0 0 1	ADD (LFX)
0 0 0 0 1 0		1 0 0 0 1 0	ADD (FLT)
0 0 0 0 1 1		1 0 0 0 1 1	ADD (DEC)
0 0 0 1 0 0	CVL (SFX)	1 0 0 1 0 0	
0 0 0 1 0 1	NOP	1 0 0 1 0 1	
0 0 0 1 1 0	CVL (FLT)	1 0 0 1 1 0	
0 0 0 1 1 1	CVL (DEC)	1 0 0 1 1 1	
0 0 1 0 0 0	CVF (SFX)	1 0 1 0 0 0	SUB (SFX)
0 0 1 0 0 1	CVF (LFX)	1 0 1 0 0 1	SUB (LFX)
0 0 1 0 1 0	NOP	1 0 1 0 1 0	SUB (FLT)
0 0 1 0 1 1	CVF (DEC)	1 0 1 0 1 1	SUB (DEC)
0 0 1 1 0 0	CVD (SFX)	1 0 1 1 0 0	CPRA (SFX)
0 0 1 1 0 1	CVD (LFX)	1 0 1 1 0 1	CPRA (LFX)
0 0 1 1 1 0	CVD (FLT)	1 0 1 1 1 0	CPRA (FLT)
0 0 1 1 1 1	NOP	1 0 1 1 1 1	CPRA (DEC)
0 1 0 0 0 0	NEG (SFX)	1 1 0 0 0 0	MPY (SFX)
0 1 0 0 0 1	NEG (LFX)	1 1 0 0 0 1	MPY (LFX)
0 1 0 0 1 0	NEG (FLT)	1 1 0 0 1 0	MPY (FLT)
0 1 0 0 1 1	NEG (DEC)	1 1 0 0 1 1	MPY (DEC)
0 1 0 1 0 0	ABS (SFX)	1 1 0 1 0 0	POLY (SFX)
0 1 0 1 0 1	ABS (LFX)	1 1 0 1 0 1	POLY (LFX)
0 1 0 1 1 0	ABS (FLT)	1 1 0 1 1 0	POLY (FLT)
0 1 0 1 1 1	ABS (DEC)	1 1 0 1 1 1	POLY (DEC)
0 1 1 0 0 0	MNS (SFX)	1 1 1 0 0 0	DIV (SFX)
0 1 1 0 0 1	MNS (LFX)	1 1 1 0 0 1	DIV (LFX)
0 1 1 0 1 0	MNS (FLT)	1 1 1 0 1 0	DIV (FLT)
0 1 1 0 1 1	MNS (DEC)	1 1 1 0 1 1	DIV (DEC)
0 1 1 1 0 0	TA (SFX)	1 1 1 1 0 0	
0 1 1 1 0 1	TA (LFX)	1 1 1 1 0 1	
0 1 1 1 1 0	TA (FLT)	1 1 1 1 1 0	
0 1 1 1 1 1	TA (DEC)	1 1 1 1 1 1	

Figure 4.1.3.2/1 continued Instruction Mnemonic Codes

## PAU INSTRUCTION CODES



MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION	MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION
0 0 0 0 0 0	RESTART	1 0 0 0 0 0	AREA
0 0 0 0 0 1	RESUME	1 0 0 0 0 1	LIST
0 0 0 0 1 0	BOOLE	1 0 0 0 1 0	LISTSZ
0 0 0 0 1 1	SETORG	1 0 0 0 1 1	LISTLZ
0 0 0 1 0 0		1 0 0 1 0 0	REPLICATE
0 0 0 1 0 1		1 0 0 1 0 1	RDERLZ
0 0 0 1 1 0		1 0 0 1 1 0	WRITLZ
0 0 0 1 1 1		1 0 0 1 1 1	WRERLZ
0 0 1 0 0 0	GATEIA	1 0 1 0 0 0	TESTP
0 0 1 0 0 1		1 0 1 0 0 1	TESTB
0 0 1 0 1 0		1 0 1 0 1 0	LISTI
0 0 1 0 1 1		1 0 1 0 1 1	ERASEP
0 0 1 1 0 0		1 0 1 1 0 0	SHIFT
0 0 1 1 0 1		1 0 1 1 0 1	TALLY
0 0 1 1 1 0		1 0 1 1 1 0	TALLYHO
0 0 1 1 1 1		1 0 1 1 1 1	CONNECT
0 1 0 0 0 0	LOADB	1 1 0 0 0 0	CLEARP
0 1 0 0 0 1	PUSHB	1 1 0 0 0 1	SETP
0 1 0 0 1 0	STOREB	1 1 0 0 1 0	MOVEB
0 1 0 0 1 1	POPB	1 1 0 0 1 1	READLZ
0 1 0 1 0 0		1 1 0 1 0 0	PLOT
0 1 0 1 0 1		1 1 0 1 0 1	PLOTSZ
0 1 0 1 1 0		1 1 0 1 1 0	PLOTLZ
0 1 0 1 1 1		1 1 0 1 1 1	PLOTI
0 1 1 0 0 0		1 1 1 0 0 0	COPY
0 1 1 0 0 1		1 1 1 0 0 1	COPYC
0 1 1 0 1 0		1 1 1 0 1 0	PLAND
0 1 1 0 1 1		1 1 1 0 1 1	PLOR
0 1 1 1 0 0	TOPOLOGY	1 1 1 1 0 0	PLNAND
0 1 1 1 0 1	TOPOLOGY	1 1 1 1 0 1	PLNOR
0 1 1 1 1 0	TOPOLOGY	1 1 1 1 1 0	PLEXOR
0 1 1 1 1 1	TOPOLOGY	1 1 1 1 1 1	PLEQV

Figure 4.1.3.2/1 continued Instruction Mnemonic Codes

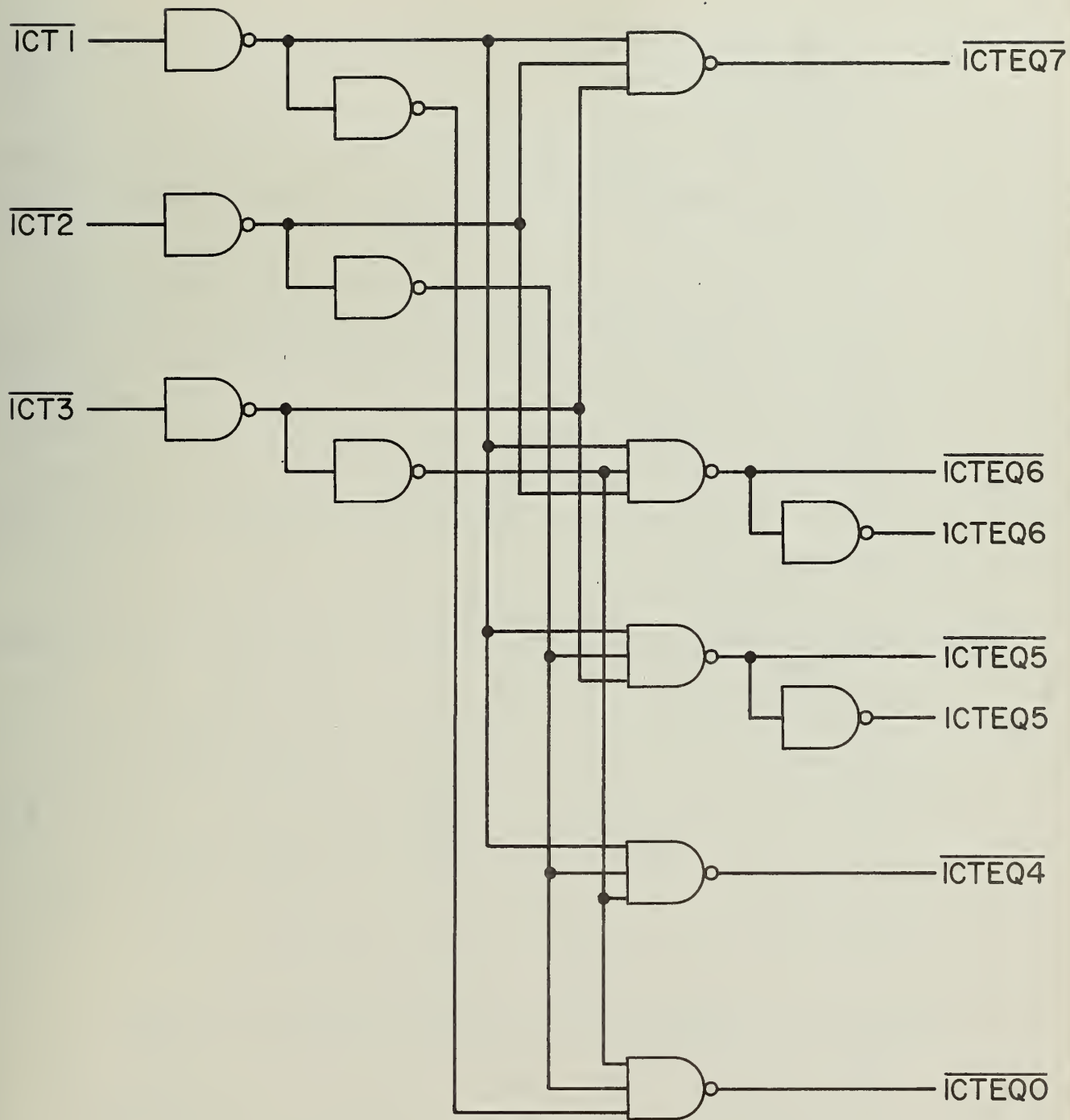


Figure 4.1.3.2/2

ICT Decoder



After control point PIT1, a decoding network is needed to determine if the IBR must be reloaded. The decision structure is shown in the flowchart at the end of Section 4.1.3.2 and the logic used to implement this structure is shown in Figure 4.1.3.2/3. Note that since time is not important for this circuit, a minimum logic design was used even though it takes 4 collector delays.

Control point PIT2 and the string of control points PIT4 through PIT8 constitute another interesting situation. If an interrupt occurs during the call by PIT2 to the IBR Reload sequence, it is still necessary to restore the ICT to its original value. It is necessary, therefore, to save the outcome of the IBR Reload sequence by setting a flip-flop. If PIT3 is selected instead of PIT2, the flip-flop must be reset so that no interrupt return is made. The actual decision is made following control point PIT8.

PIT11, PIT12, and PIT13 constitute a loop which may be executed twice if a three operand phrase instruction is being processed. A flip-flop, which is initially reset by control point PIT9, is used to direct the flow of control during the execution of the loop. If the instruction is a three operand one, then, after the first execution of the loop control point PIT14 is activated. This control point is not shown in the control point flow chart at the end of this section since all it does is set the control flip-flop. As soon as this is done control is returned to the beginning of the loop. After the second execution control is given to control point PIT15.

During the execution of the loop, an interrupt may occur in the Phrase Sequence. If so, the control flip-flop is used to select which Interrupt Sequence entry point to activate. Eventually, when the interrupt has been taken care of and control is returned via one of the two interrupt return points, the control flip-flop must be set to its original state right before the interrupt was detected.

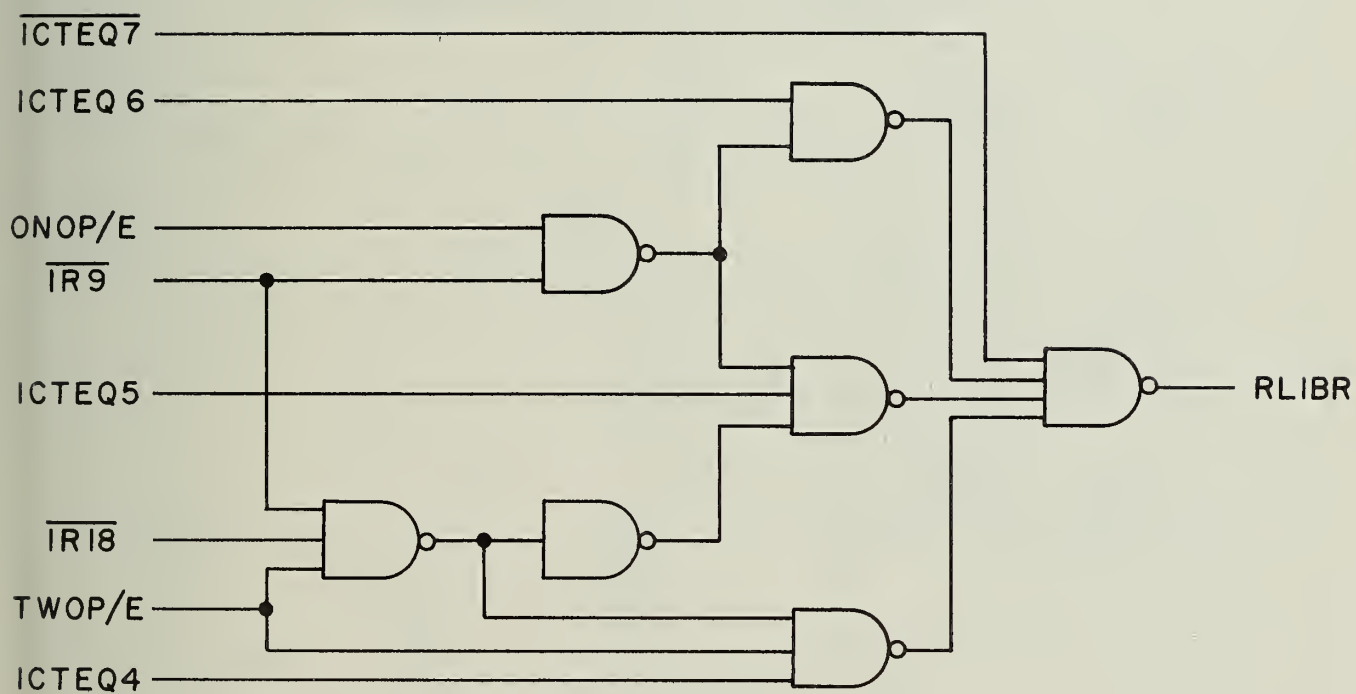
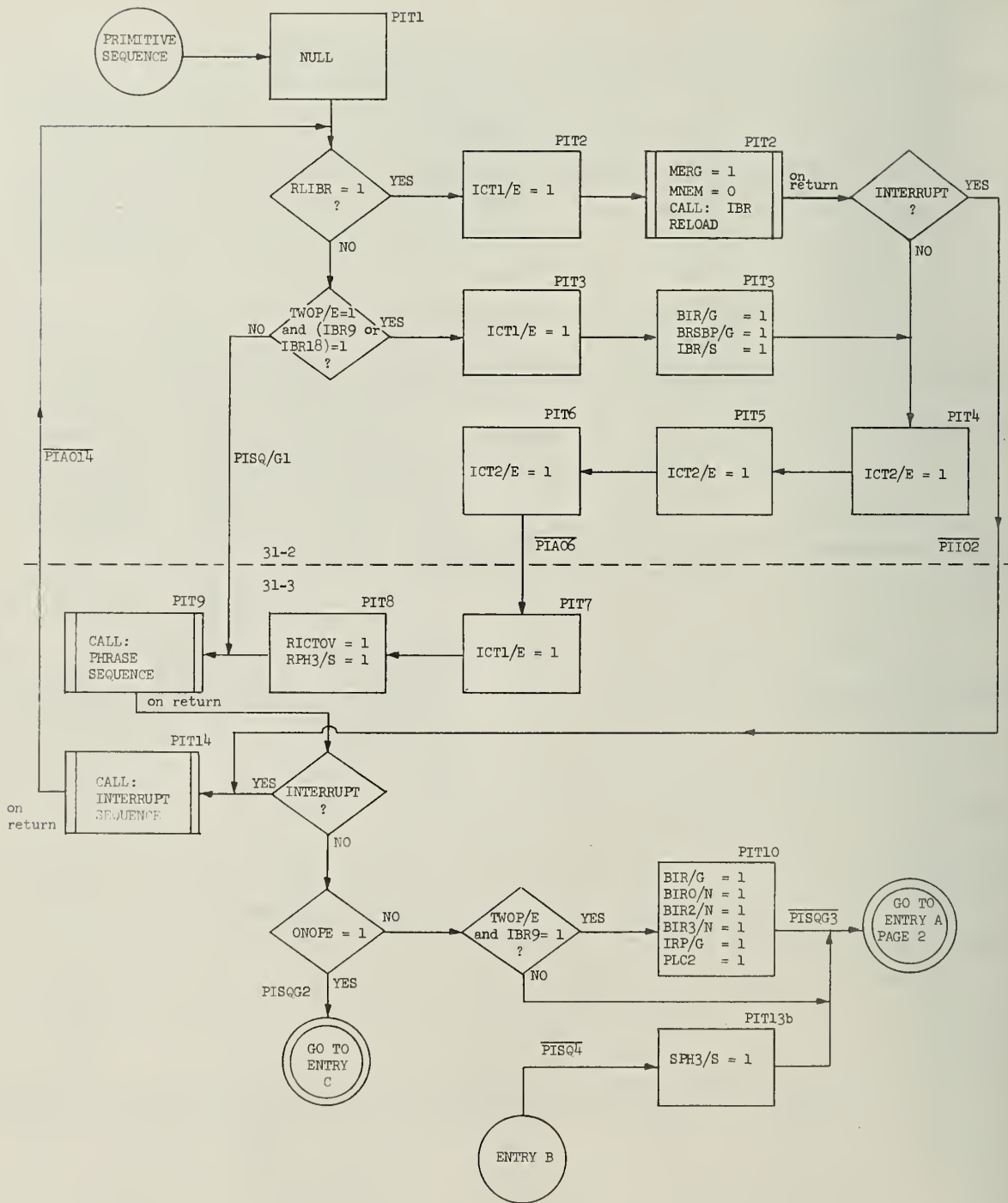


Figure 4.1.3.2/3

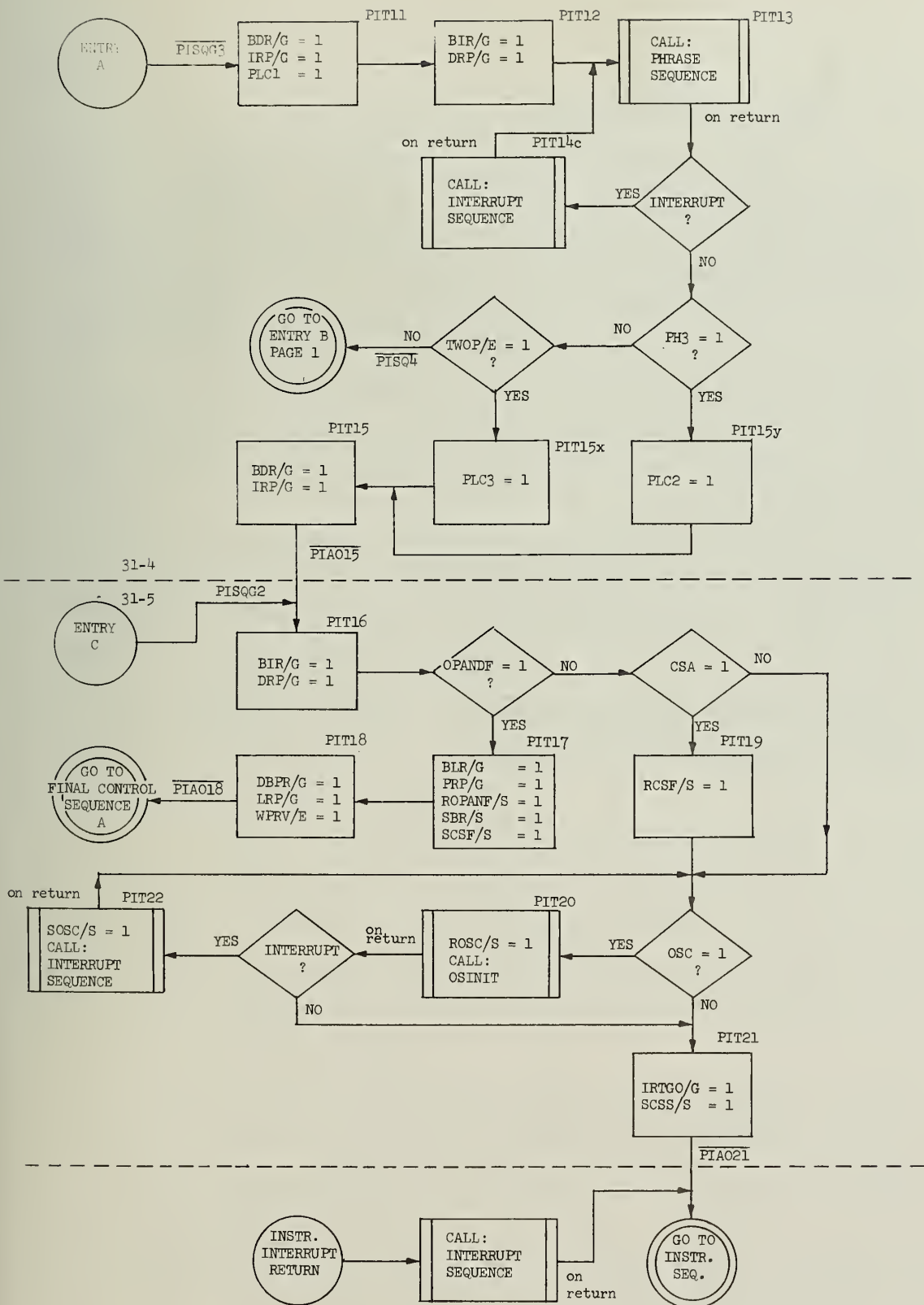
Circuit to Determine Reloading of IBR During Prim. Inst. Sequence

10/30/69



Primitive Instruction Sequence -  
Initial Control Sequence Control Step Flow Chart





Primitive Instruction Sequence -  
Initial Control Sequence Control Step Flow Chart

#### 4.1.4 Final Control Sequence

After the execution of an instruction sequence, control is given to the Final Control Sequence. During this sequence the post execution "pops" are performed on the operand phrases, if necessary, and the instruction counter is incremented. At this time various checks are made for PR#0 modification, changes in the operand stack pointer, an empty instruction buffer register, interrupt conditions, or the end of an execute instruction. After all of these conditions have been examined, and taken care of if necessary, control is given to the beginning of the Main Control Sequence and a new instruction cycle begins.

The following sections will give a detailed description of the operation and the control logic for the Final Control Sequence. In Section 4.1.4.1 the operation will be described in two distinct phases, just as in the Primitive sequence, so that the effects of the interrupts may be described separately. This will hopefully make the sequence operation less confusing.

#### 4.1.4.1 Final Control Sequence Description

The Final Control Sequence has two entry points, entry A which is the normal entry point for primitive instructions and entry B which is used to skip around the operand phrase post-operations and the incrementing of the instruction counter. This latter entry point is used by the Imprimitive instructions.

The first part of the sequence utilizes the same logic used in the Primitive Sequence to classify the instruction as to the number of operands it has. This information is then used to determine how much to increment the ICT in order to point to the next instruction. Note however that the post-slash operations are performed on all the operand phrases before the ICT is incremented. This insures that a proper value is obtained if PR#0 is used in an operand phrase and is popped. If PROMØD = 1 the ICT is not incremented.

Care must be taken when using PR#0 as an operand phrase which is to be modified or popped. In order to ensure predicable results the following facts should be considered:

- 1) PR#0 is incremented after instruction execution and after all the operand post-slash operations have been performed.
- 2) If PR#0 is used in a primitive operand phrase and is modified by the phrase operation or the instruction itself, PROMØD will be set to 1 by the time the final control sequence begins.
- 3) If a pre - and post-slash are both specified on PR#0 when it is used as an operand in a primitive instruction, PROMØD will be set to 0 after the post-slash has been executed.
- 4) If a post-slash on PR#0 is specified by a primitive instruction operand phrase with no pre-slash, PROMØD is set to 1 after execution of the post-slash.

After the ICT has been incremented entry B joins the sequence. A check is made of OSC. If it is "1" the OS has been cleared during the execution of the primitive instruction and the stack must be reinitialized.

If PROMOD = 1, then PR#0 has been modified during the instruction and the next instruction will not occur in sequence. Therefore a memory access to the new address must be made and the IBR loaded with the instruction at this point.

The IBR will also have to be reloaded if the ICT = 0 since this means that the instruction buffer is empty.

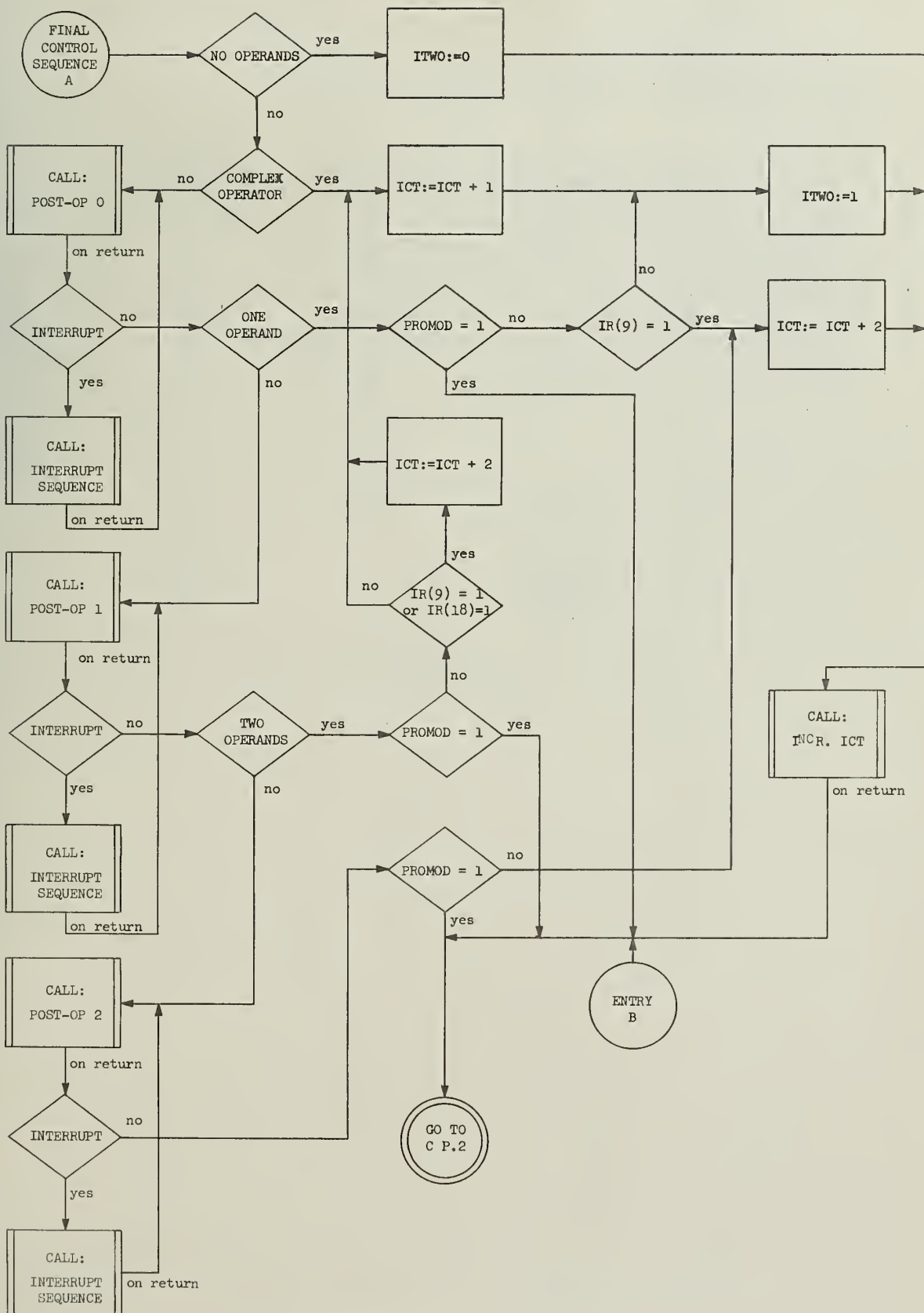
Next the DR is loaded with the contents of PR#0 and PROMOD and OSC are reset to zero after which a check is made for any external interrupts which may have occurred since the last time an external interrupt check was made.

Finally bit 9 of the DR is checked to see if an EXECUTE instruction is being processed. If so, return is made to entry B of the Imprimitive Sequence. Otherwise, the control begins again at the Main Control Entry.

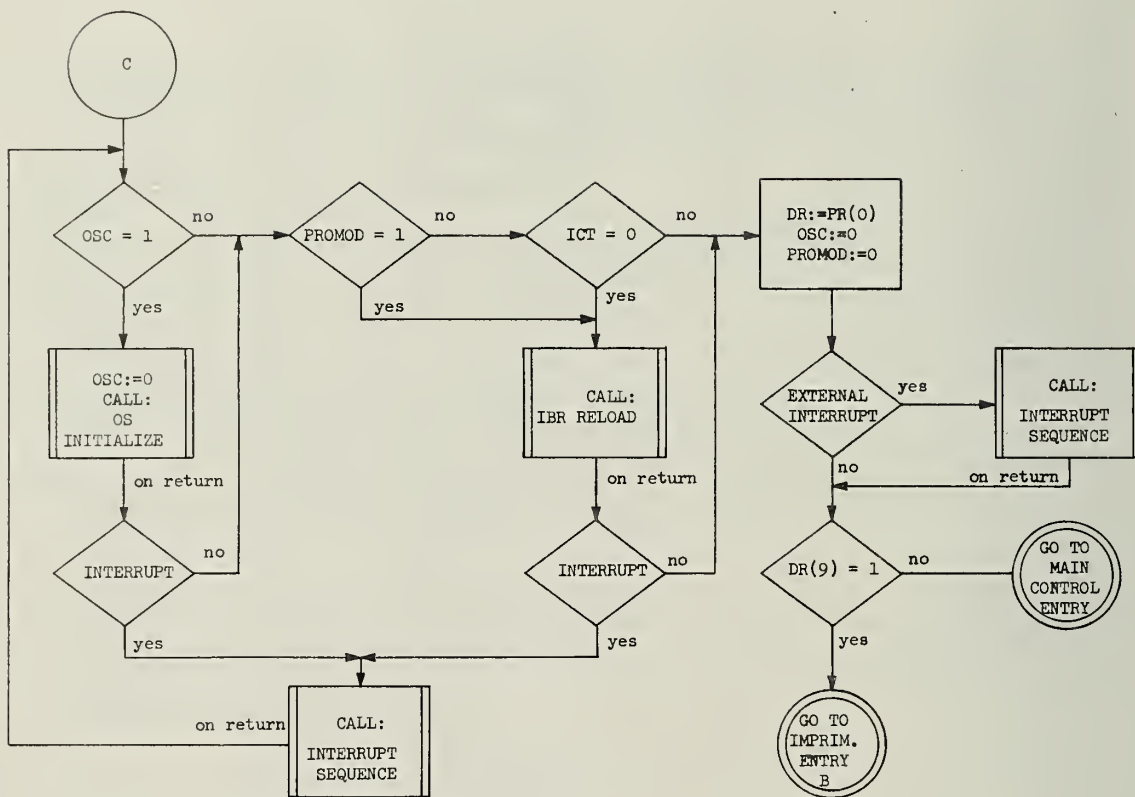
As in the Primitive sequence the treatment of interrupts in the Final Control sequence is complicated by the fact that the sequence is not necessarily restartable if an interrupt occurs in one of the sequences called by the sequence. There are actually four possible interrupt return starting points, labeled E, F, G, and H.

If an interrupt occurs during the operation of the POST OP 0, 1 or 2 sequences the interrupt must return to interrupt return points E, F or G respectively. This is because once a post-op sequence has been performed successfully it cannot be redone without screwing everything up and since the old PR contents has been destroyed, there is no way to undo it.

If an interrupt occurs during the OS INITIALIZE sequence or the IBR RELOAD sequence the interrupt will return to interrupt return point 1



FINAL CONTROL Sequence Flow Chart



FINAL CONTROL Sequence Flow Chart



#### 4.1.4.2 FINAL Control Logic

The Final Control sequence completes the execution of the instruction and tests for interrupts and the completion of EXECUTE instructions.

The decision logic which appears after control points FCT3, FCT4 and FCT5 is fairly complex. As can be noted from the flow charts, these control points can possibly activate control points FCT2, FCT6, FCT7, FCT12, FCT13 or FCT14. Figure 4.1.4.2/1 lists in boolean form the functions which will activate each of these control points. The FCA0 signals shown indicate the Advance Out,  $A_0$ , signal from the corresponding control points.

FCT2:

FCA01 .  $\overline{\text{OP0}}$  . OPC v  
FCA05 v  
FCA03 . OP1 v  
FCA04 . OP2 .  $\text{PROMOD}(\overline{\text{OP1}} \cdot \overline{\text{OP2}} \vee \text{OP2}(\overline{\text{IBR9}} \cdot \overline{\text{IBR18}}))$

FCT6:

FCA05 v  
FCA03 . OP1 v  
FCA04 . OP2 .  $\overline{\text{PROMOD}} \cdot \text{OP1} \cdot \text{IBR9}$

FCT7:

FCA03 . OP1 v  
FCA04 . OP2 v  
FCA05 .  $\overline{\text{PROMOD}} \cdot \overline{\text{OP1}} \cdot \text{OP2}(\text{IBR9} + \text{IBR18})$

FCT12:

ENTRYB . OSC v  
FCA016 . OSC v  
FCA03 . OSC . PROMOD . OP1 v  
FCA04 . OSC . PROMOD . OP2 v  
FCA05 . OSC . PROMOD

FCT13:

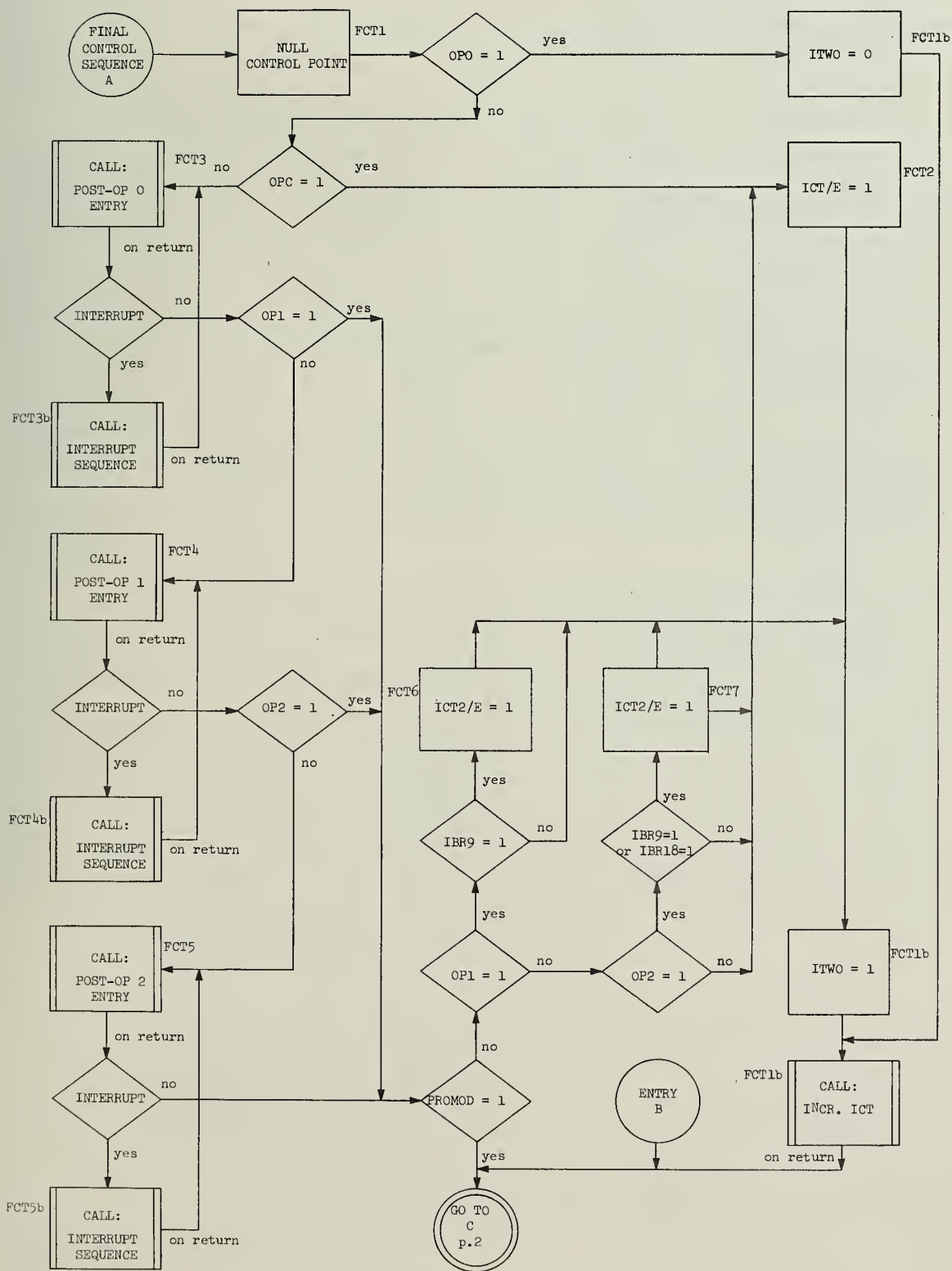
ENTRYB . OSC v  
FCA016 . OSC v  
FCA03 . OSC . PROMOD . OP1 v  
FCA04 . OSC . PROMOD . OP2 v  
FCA05 . OSC . PROMOD v  
FCA012 .  $(\text{PROMOD} \vee \overline{\text{PROMOD}} \cdot \text{ICTEQ})$

FCT14:

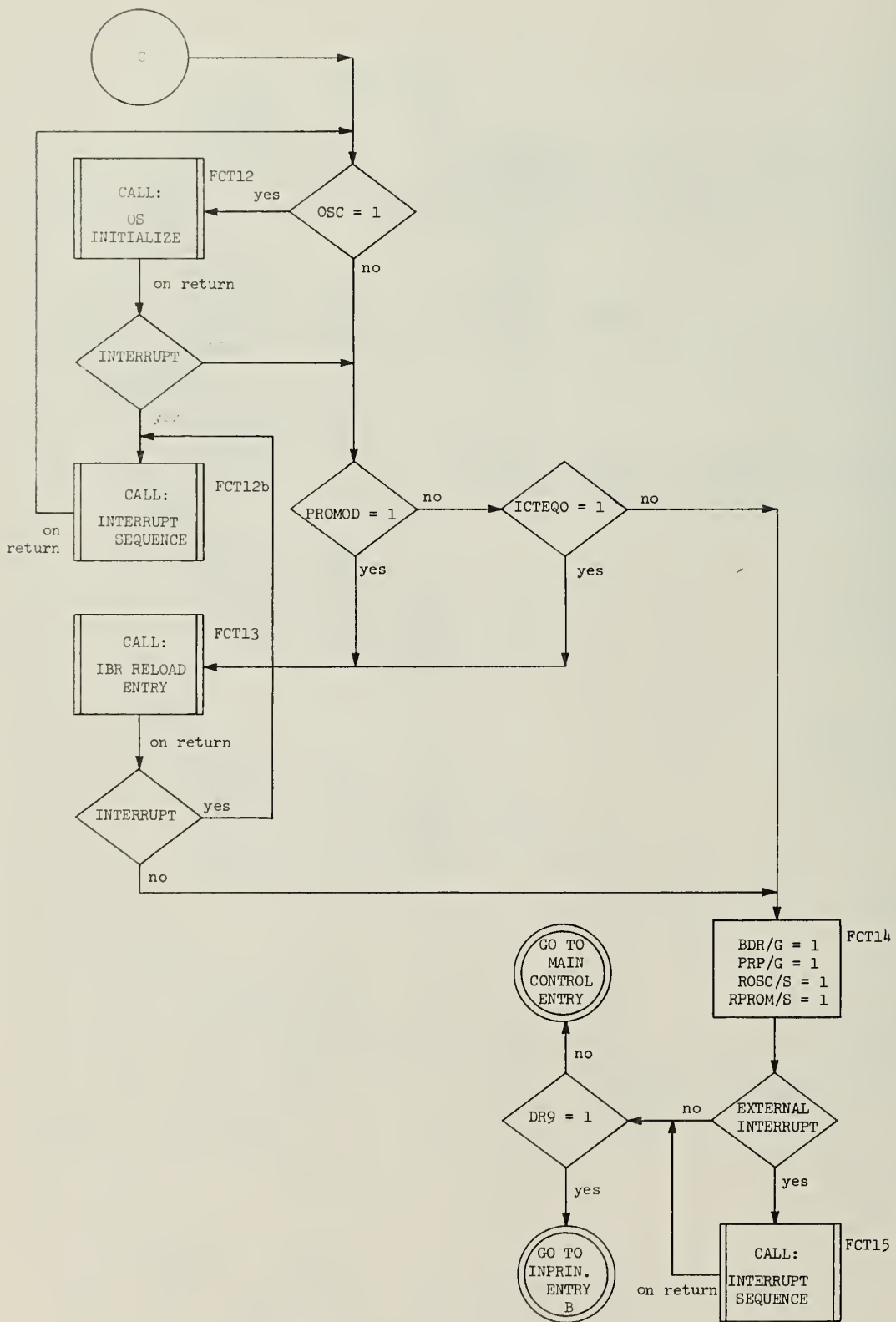
ENTRYB . OSC v  
FCA016 . OSC v  
FCA03 . OSC . PROMOD . OP1 v  
FCA04 . OSC . PROMOD . OP2 v FCA05 . OSC . PROMOD  
FCA012 .  $(\text{PROMOD} \vee \overline{\text{PROMOD}} \cdot \text{ICTEQ}) \vee$   
FCA013 .  $\overline{\text{PROMOD}} \cdot \overline{\text{ICTEQ}}$

Figure 4.1.4.2/1 Boolean Functions for  
Control Points FCT2, FCT6, FCT7, FCT12, FCT13





FINAL CONTROL Sequence Control Step Flow Chart



FINAL CONTROL Sequence Control Step Flow Chart

## 4.2 Memory Access

The Memory Access Sequence is the most complicated basic sequence in the Taxicrinic Processor.

The main task of the Memory Access Sequence is to obtain data. The data may be of the immediate type, in which case no access to core is necessary, or it may be non-immediate, in which case one or more accesses to core will be needed. If access to core is required, the given virtual address is automatically checked against the contents of the bounds field of the base register. The alignment of the cell with boundaries appropriate for the given cell size is also checked. The address in core is then constructed relative to the base address. If the base information for the segment name being requested is not in one of the seven "active" base registers, the Memory Sequence will automatically access the Segment Name Table, load the proper base information into the least-used base register and continue with the access.

There are two distinct memory organizations, both of which are hardware implemented. In the Contiguous Data Access Mode all memory accesses are made through the pointer registers directly to the user's file by means of the base information. The Partitioned Data Access Mode makes use of a page map (in memory) which is accessed first to identify the page in memory which is to be accessed. This procedure allows for a more flexible storage organization at the cost of a slower access time. Both the Segment Name Table and the segments themselves may be partitioned.

#### 4.2.1 Modes of Memory Access

##### 4.2.1.1 Contiguous/Partitioned Storage Organization

There are two types of memory organization in the Illiac III system: contiguous and partitioned. As far as the user is concerned virtual memory is conceived as contiguous, independent of the selected mode of memory organization.

In the Contiguous mode each segment is referenced by its base information contained in a base register. This information consists of the page address of the first page in the segment, the length of the segment and the access privilege of the segment. Each page in the segment must be contiguous with the rest of the segment and all of the pages have the same access privilege.

In the Partitioned mode the various pages of the segment do not have to be contiguous. Pages can be stored in distinct areas of memory (possibly units having different access times), with each page having its individual access privilege.

The Partitioned mode provides several advantages in storage allocation:

- 1) It is much easier to find storage for large files since the files do not have to be contiguous.
- 2) There is never a need for a complex "garbage collection" of blocks of storage which have been discarded by various processes, since there is no need to try to combine them into the largest contiguous blocks possible.
- 3) If a file, at some time subsequent to its initial storage allocation, requests more space, it can be conveniently taken from any unassigned pages in storage.

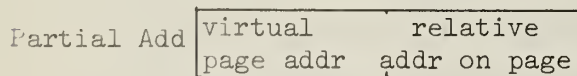
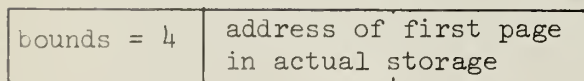
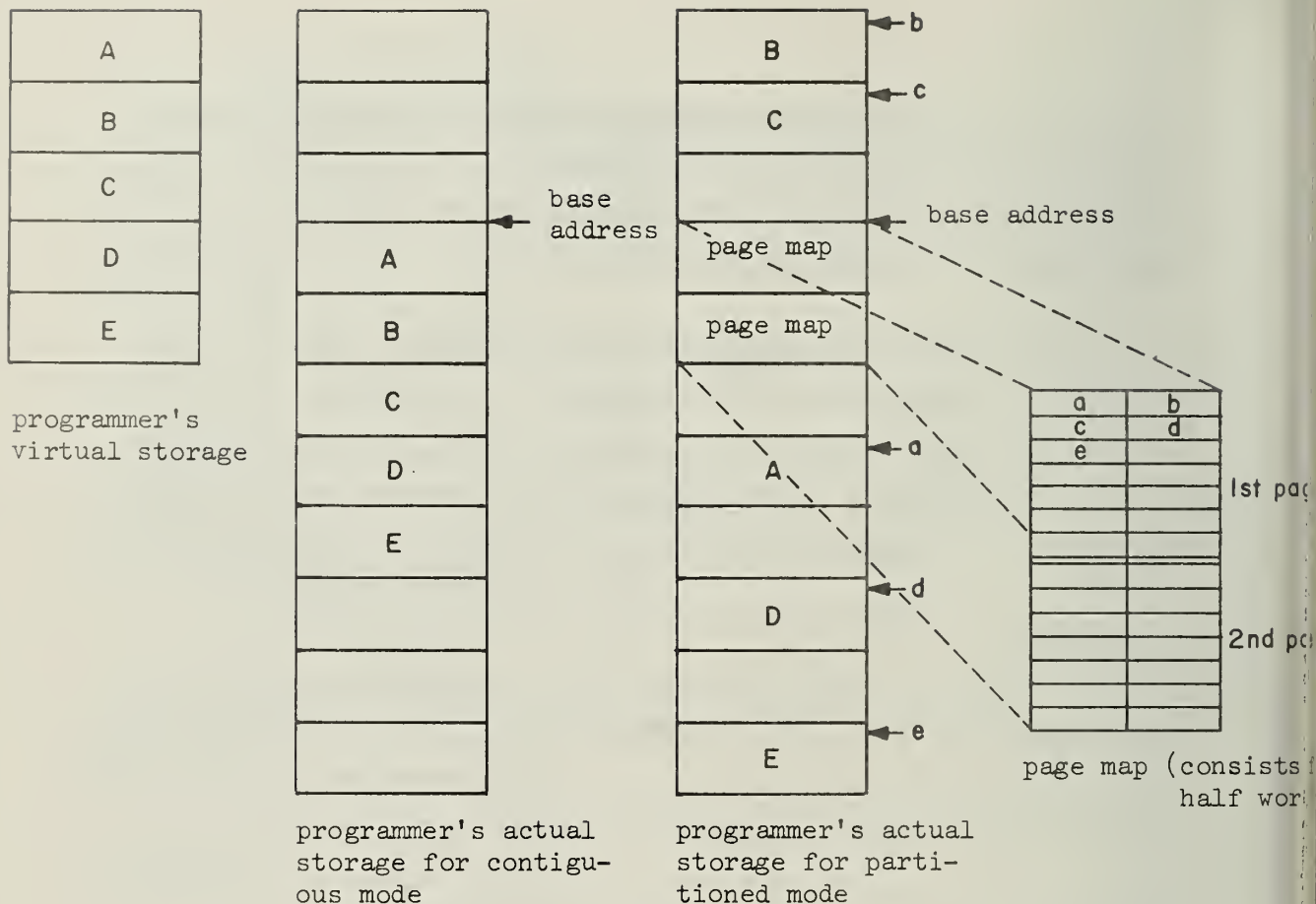
The basic idea behind the partitioned mode is to take more efficient advantage of storage by discarding the need for contiguous storage. To allow a file to maintain storage pages in an arbitrary number of places in memory (up to 256 pages) however, a strategy is

required to keep track of these page locations. The memory access strategy should demand a minimum of effort on the programmer's part, and as little extra hardware and time overhead as possible. To these ends the following decisions were made:

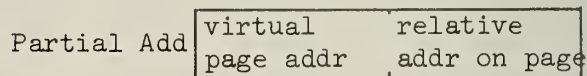
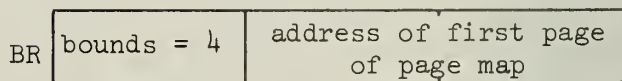
- 1) The implementation of the Partitioned Mode of memory accessing will be completely in hardware.
- 2) File storage allocation will be done by the Operating System. The programmer need not worry about the mode of memory access unless he specifically desires to request one or the other modes of operation.

To keep track of the pages in the segment, a page map is used. This consists of two consecutive pages in memory which are set up by the Operating System when the user first requests storage allocation. The map contains up to 256 half-words which specify the page addresses in actual storage of the user's virtual memory pages and for each page, its access mode. These 16-bit half-words are arranged in the order in which the user's pages appear in virtual memory. (See Figure 4.2.1.1). In Partitioned Mode the base information is used to indicate the location of the page map and its access mode. In common with the Contiguous Mode the bounds field of the base register is used to make sure the address is within the virtual memory assigned to the segment. Each time an access is desired, the control unit checks the first flag bit in the base register for a "1" to see if the Partitioned Mode is to be used. If it is, the page address field of the virtual address (or file address), i.e., the page address in the user's virtual memory, is converted to an address of a half-word in the page map. This half-word contains the address in actual memory of that particular page





Contiguous Format



Partitioned Format

Figure 4.2.1.1 - Pictorial Comparison of Partitioned Mode vs. Contiguous Mode of Data Accessing

Note: Virtual Addresses (those constructed by programmer) are the same. Only BR's are different and these should only be assigned by the Operating System. Programmer will not know the difference.

in the user's virtual storage. After its page location has been accessed, the complete address of the data desired can be obtained by combining the page address with the relative byte address on the page as given in the original virtual address.

#### 4.2.1.2 Memory Sequence Entry Points

There are two possible entry points to the memory sequence, the Direct and the Modified entry. In the Direct entry the eventual address is determined by the particular Pointer Register specified by the TGR. This entry is used for most operands which are specified by Pointer Registers.

The Modified Entry is used by internal sequences which need to "manufacture" their own addresses. It cannot have the immediate address option. In the case where the Modified Entry is used the DR must contain the constructed address and the selection of the PRSN register must have already been completed.

A LOAD-type operation is a single entry-exit operation. When the exit is made, the required operand cell is found left-justified in the LR (in the case of a byte, halfword, or word) or it is in the LR and the DR (in the case of a double word).

A STORE-type instruction is a dual entry-exit operation: The first entry is made for one of the two possible entry variants; the first exit is made back to the main control from the store sequencing when the address has been constructed and checked. At this point, the data cells to be stored are loaded into the LR (and the DR in the case of a double word), left justified. The second entry to the store sequencing is then done, and the data cell(s) are stored in the operand address as requested. At the second exit, the store operation is complete.



#### 4.2.2 Constituent Tasks of Memory Access

The purpose of this section is to describe the various parts of the memory sequence in considerable detail. The detailed logical design will be given as well as its functional operation. The following section (section 4.2.3) will give a description of the entire memory sequence operational flow chart. The final section (section 4.2.4) describes the control logic used in the memory sequence and its control point flow chart.

#### 4.2.2.1 Cell Alignment Check

The TP Memory sequence requires that all cells which are accessed in core be aligned in such a way that a cell of a given size does not cross any correspondingly sized boundary in core. Thus a double word cell must not cross a double word boundary in core, a word cell must not cross a word boundary in core, etc. This restriction greatly eases the burden which would fall on the TP if arbitrarily located cells had to be accommodated. Also since the two major data streams which would require arbitrary boundaries, i.e. the OS and the instruction stream, are automatically taken care of with double word buffers, there is no great hardship on the programmer. It is hoped that with the experience of the 360 systems programming, the problem of writing compilers and assemblers which will assign cells to their proper boundaries will not be too great.

At any rate it is the responsibility of the memory sequence to check every access request to ensure that the cell is properly aligned. This is a simple matter since it only involves checking the cell size and the low order 3 bits of the address. This is done at the beginning of the memory sequence before the segment name is compared with the association logic to see if the base information is in one of the BR's. The logic to detect a misaligned cell is shown in Figure 4.2.2.1/1.

If a misalignment is discovered, an alignment check interrupt is generated.

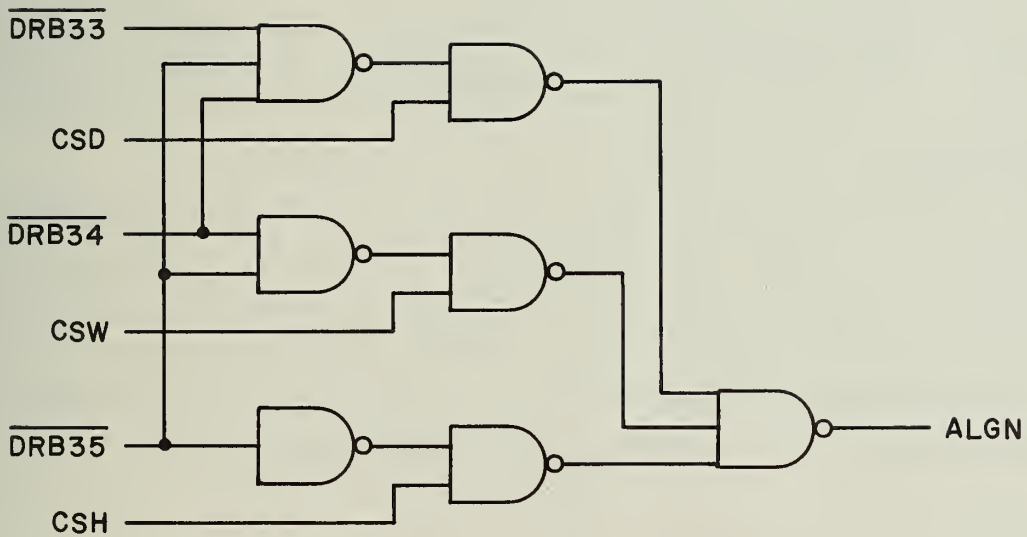


Figure 4.2.2.1/1 Alignment Check Logic

#### 4.2.2.2 Base Register Check

At the beginning of a memory access a check must be made that the necessary base information is present in one of the hardware base registers in the TP. This is done by the Association Logic described in connection with the Base Registers in Section 2.5.1.4. If no base register is found with the required base information, the Memory Sequence must access the Segment Name Table to obtain the base descriptor and then load this data into the base register which has been used least recently (see Section 2.5.1.3 for a description of this selection process). It should be noted that the Segment Name Table may be partitioned and thus it may be necessary to go through a page table to get to the Segment Name Table.

### 4.2.2.3 Address Bounds Check

#### 4.2.2.3.1 Address Bounds Check - Functional Description

The purpose of this logic is to examine all outgoing memory addresses to insure that an address falls within the area allotted to the segment and hence, program, in question.

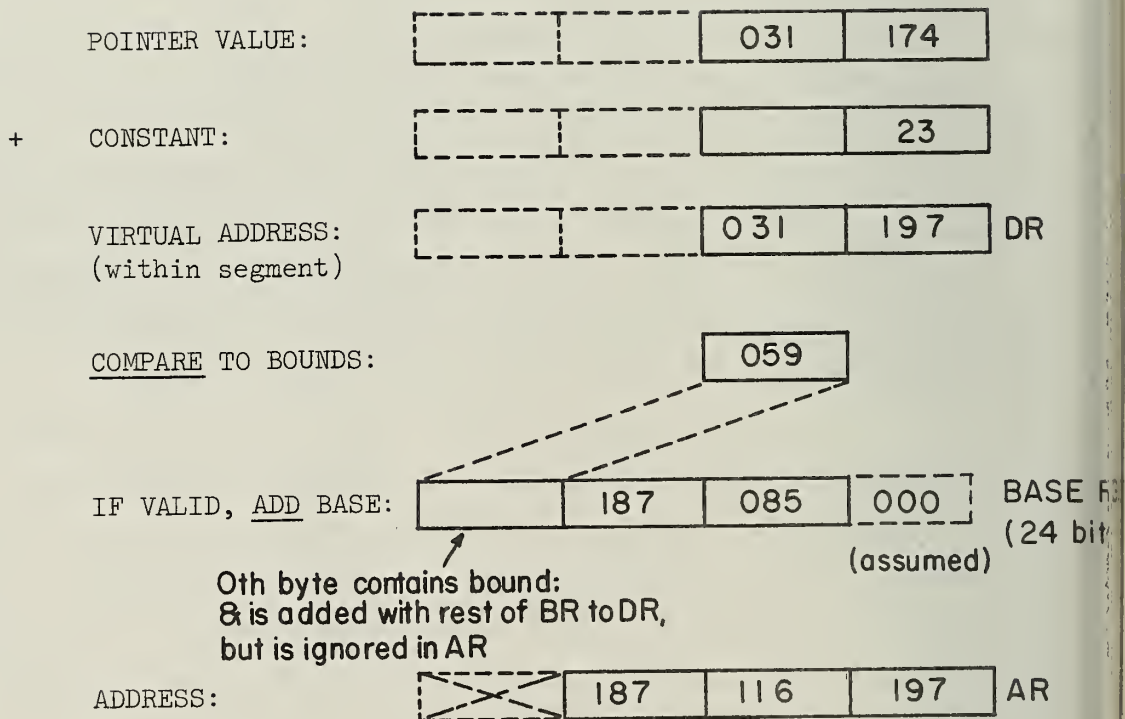
An address is constructed using the base address and either a pointer value field (direct entry) or a preformed constant (modified entry). If the pointer value or constant passes a comparison with the given bounds for that segment, it can be added to the base address (the high order 16 bits of a 24 bit address) to give a final 24 bit address which may then be sent to the core unit. (This is the procedure for a contiguous mode access; partitioned mode will be explained later in Section 4.2.2.5). A graphical illustration of the process is given in Figure 4.2.2.3.1.

The base registers hold a 2-byte (16 bit) base PAGE ADDRESS and an associated 1-byte page count\*, or BOUNDS. The page address indicates the beginning of the segment associated with the given base register. The BOUNDS is one less than the number of pages in the segment. This greatly simplifies checking the logic for a relative overflow (see below). The minimum allowed segment length is thus one page, corresponding to a BOUNDS = 0. The maximum is 256 pages (BOUNDS = 255). One page contains 256 bytes, 64 words, or 32 double words.

In the bounds check, the 16-bit virtual address is compared with the given BOUNDS to check for an "absolute" bounds overflow, i.e. the virtual address exceeds the page count.

---

\*Actually (page count) - 1; see text.



Note: This file is allowed 60 pages of storage beginning at page 

187	085
-----	-----

. For this example, the numbers in each byte are the decimal representation of the binary contents, i.e. they run from 0 to 255.

Figure 4.2.2.3.1 - Address Bounds Check

If this condition occurs, the bounds overflow indicator (BOV) is set and an interrupt may be initiated. As the actual bounds checking begins, the DR contains the virtual address while the DB holds the bounds. The BOUNDS byte (which is the leftmost byte in the DB) and the page address byte of the virtual address (the second byte from the right in the DR) are simultaneously compared for  $DR > DB$ . If this does not occur the address is within bounds. If it does occur, an absolute overflow will take place.

In the case where  $DR = DB$  there is no need to worry about a bounds overflow condition since cells being accessed through the memory sequence are not allowed to cross double word boundaries. Thus any cell which began in the last page assigned to the segment and extended over the boundary would be illegal simply because of improper cell alignment.

For memory access in the Partitioned Mode the bounds check still works satisfactorily. In this case the bounds check method for the Contiguous Mode determines if the user is trying to exceed his allotted virtual storage. Since the user is not allowed to change his page map, there is no danger of the user straying into someone else's storage as long as he remains inside any one of his pages. At the same time, if the user tries to access data which overlaps a virtual page boundary, he will again be stopped because of improper cell alignment.



#### 4.2.2.3.2 Address Bounds Check-Logic Description

The purpose of the address bounds check logic is to rapidly check the 8 bit page number of the virtual address which will be accessed, against the 8 bit bounds which is contained in the base register bounds field. In order to do this rapidly, the comparison is done in parallel fashion without using the adder.

The actual bounds checking logic is given in the TP Logic Book in Drawings 06-01 and 06-2. The eventual purpose of the logic is to set the BOVI (bounds overflow) flip-flop if this is necessary. This flip-flop is composed of cross-connected NAND's on 240-00 cards, as shown in Figure 4.2.2.3.2/1. When not in use the input signals to this flip-flop rest at "1". Initially the flip-flop can be reset by setting the reset system signal,  $\overline{\text{RSYS}}$ , to "0".

At the beginning of the bounds checking operation,  $\overline{\text{BDCK/E}}$  goes to "0" and BDCK/E goes to "1" (the timing is not important in this case). Since BDCK/E is common to every input circuit in the checking logic this effectively begins the checking operation. At this time the DR contains the address to be checked while the DB contains the bounds and the base address. In particular bits 19 through 26 of the DR contain the page portion of the address, while bits 1 through 8 of the DB contain the bounds.

In order to set the flip-flop properly the logic in Drawing 06-1 is used to generate the  $\overline{\text{BGT}}$  signal, which if off indicates that the number represented by the DR bits is greater than the number represented by the DB bits. The logical equation for this signal is as follows:

$$\text{BGT} = \overline{\text{DB}(1)} \cdot \text{DR}(19) \vee \overline{\text{DB}(2)} \cdot \text{DR}(20) \cdot \overline{\text{I}(2)} \vee \dots \\ \vee \overline{\text{DB}(8)} \cdot \text{DR}(26) \cdot \overline{\text{I}(8)}$$

where

$$\text{I}(2) = \text{DB}(1) \cdot \overline{\text{DR}(19)}$$



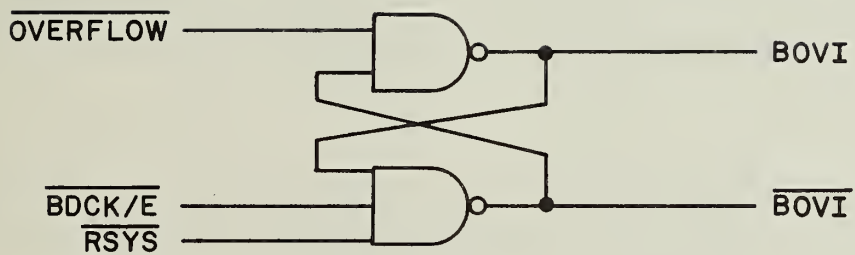


Figure 4.2.2.3.2/1 Bounds Overflow Flip-flop Logic

$$I(i) = DB(i-1) \cdot \overline{DR(18+i-1)} \cdot \overline{I(i-1)} \cdot \dots \cdot \overline{I(2)}, i = 3, 4, \dots, 8$$

BGT = 1 if in some position i,  $DR(18+i) > DB(i)$  and  $I(i) = 0$

$I(i) = 1$  if in some position,  $k < i$ ,  $DB(k) > DR(18+k)$

The equation for BGT is somewhat complicated. The idea is to start at the highest order bit positions of the two numbers and scan down each position until one comes to a position in which the DR bit is a "1" and the DB bit is a "0". If at this point there is no higher order bit position with a DR bit of "0" and a DB bit of "1", then the DR is greater than the DB and BGT = 1.

In order to keep track of what happened in the previous positions the  $I(i)$  signals are used. These signals indicate for each position, i, whether or not any previous position had a DB bit of "1" and a DR bit of "0". Some examples of the use of these equations are given below:

DR < DB	DR = 10110110
	DB = 11010011
	$\overline{I} = 1000000$
	BGT = 00000000 = 0

DR > DB	DR = 10110110
	DB = 10011011
	$\overline{I} = 1111000$
	BGT = 001 = 1

Note in the first example that the first bit position to have a DR bit of 1 and a DB bit of 0 is the third most significant bit position. However note that the second most significant bit position has already had a DB bit of 1 and a DR bit of 0 so we need to ignore the third bit position and set BGT to 0.

In the second example the third position is again the most significant position to have a DR bit of "1" and a DB bit of 0. This time, however, there is no higher order bit position with a DB bit of 1 and a DR bit of 0 so that this time we want to set BGT to 1.

But what happens if DR = DB?

```
DR = DB:      DR = 10110110
               DB = 10110110
                $\overline{I} = 11111111$ 
               BGT = 00000000 = 0
```

In this case note that all the  $I(i)$ 's are 1. However since there is never any position with a DR of 1 and a DB of zero, the BGT signal is never set.

At this point we have everything we need to generate BGT with logic. In actuality we will generate  $\overline{BGT}$  by dot-oring eight NAND circuits, one for each bit position, and using these NAND's to check each bit position for a DR bit of 1 and a DB bit of 0. At the same time we also want to use the  $I(i)$  signals for each position to inhibit the NAND of its corresponding position if any previous position had a DR bit of 0 and a DB bit of 1. This is what is represented by the previous equation for BGT and what is shown in Logic Drawing 06-1 of the TP Logic Book.

The  $I(i)$  signals are all generated in parallel rather than recursively as shown in the equations. This allows a much faster circuit and does not add too much logic. The  $I(2)$  signal is the easiest since it only involves checking one previous position. The succeeding  $I(i)$  signals are generated using 241-00 circuits and a 234-02 diode matrix board (see Figure 4.2.2.3.2/2).

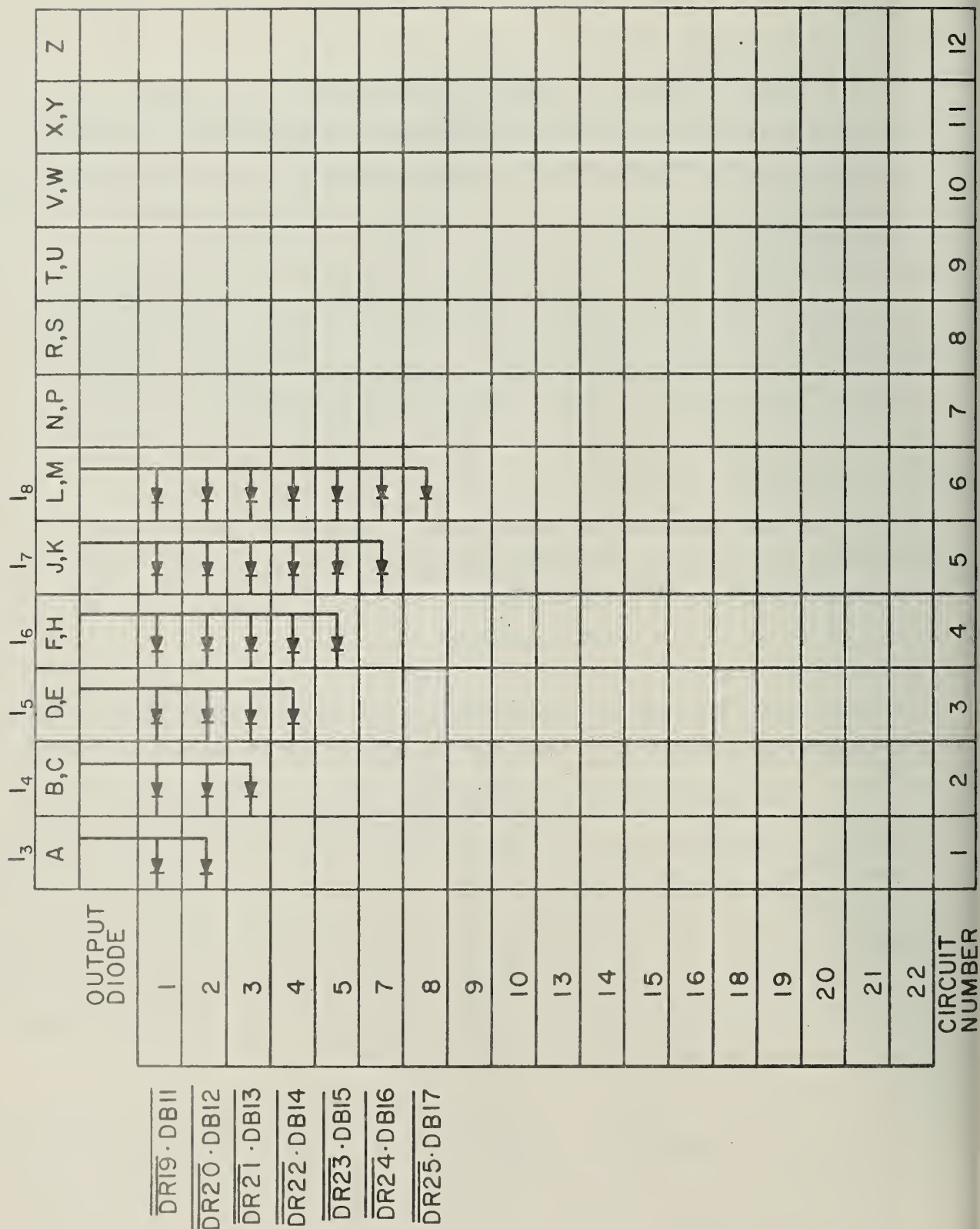


Figure 4.2.2.3.2/2 - Bounds Overflow Diode Matrix Board

Seven, three-input NAND's are used to generate a  $\overline{DR} \cdot DB$  signal for the highest order seven positions when the BDCK/E signal is turned on. These signals are then fed into the diode matrix card which AND's together increasing numbers of them and produces the I(i) output signals. If any input is a zero all the outputs using that signal will be zero and thus all the higher bit positions after the first position with a DB of 1 and a DR of 0 will be inhibited.

Once  $\overline{BGT}$  has been formed it can be used to set BOVI. If  $\overline{BGT}$  goes to zero, a bounds overflow has occurred and BOVI is set. After enough time has passed to allow for the propagation of the data from the inputs to the BOVI flip-flop, the BDCK/E signal is turned off. This will allow the flip-flop to be set if necessary since there is one extra delay before the data is removed from the set input to the flip-flop.

Figure 4.2.2.3/3 gives the signal names used by the bounds check logic.

<u>Signal Name</u>	<u>Description</u>
BDCK/E/	Bounds check enable.
BGT	Page number is greater than bounds when signal is 0.
BØV	Output signal - indicates address overflow.
CSD	Cell size is double word.
CSH	Cell size is halfword.
CSW	Cell size is word.
DBi	Permuter Output Bus (from DBP); Oth byte contains bounds during overflow check - bit i.
DRi	Distribution Register - contains virtual address during check - bit i.
RSYS/	Reset system.

Figure 4.2.2.3/3 - Signal Names Used by BØUNDS CHECK



#### 4.2.2.4 Access Privilege Bits

In order to provide file security the access privilege bits are used to identify the types of access permitted to each segment. There may be several levels of control. At each level there are always two code bits which indicate the access mode which will be allowed into the next lower level.

The access mode code is as follows:

Bit 1	Bit 2	Access Mode
0	0	Trap (no read, no write, interrupt)
0	1	Segment, or page map, page not thr
1	0	Read only - no write
1	1	Read/write allowed

The first access level is provided by the base registers and, by extension, the segment name table entries. The rightmost two flags of a base register are used for the access code. If the particular segment referred to by the base information is in contiguous mode, the access code refers to the status of the segment itself. Note that in this situation the whole segment must be loaded into core at the same time since there is no way to indicate a partial loading in contiguous mode.

If the segment is in partitioned mode then the access code refers to the status of the page map which lists the actual base addresses of each page. In this case the two flags in each base address indicate the next level of access control, i.e. the access code for each individual page of the segment.

#### 4.2.2.5 Partitioned Mode Address Construction

The initial part of the Partitioned Mode accessing sequence follows the same sequence as the Contiguous Mode. After the bounds has been checked and "approved" however, the Contiguous Mode has an address ready to be sent to core while the Partitioned Mode has only succeeded in guaranteeing that the user is not accessing outside of his virtual storage. Thus the Partitioned Mode Sequence must next construct the address which will be used to access the page map.

Referring to Figure 4.1.1 it can be seen that we want to access the n'th half word in the page map where n is the page address of the page in virtual storage which we eventually will access. This can be obtained from the Pointer Register value (which is in the DR at this point) by right shifting one byte and then left shifting one bit to get the halfword address within the page map. To this is added the base address of the page map which gives the absolute address in core of the halfword desired.

After the memory access is completed, the halfword data which was returned is used as a page address and is added to the relative address on the page (which is obtained from the PR) to generate the address of the data requested by the program. At this point the Partitioned Mode returns to the "normal access" sequence for the contiguous mode.

#### 4.2.2.6 Read/Write Byte Generator

The Read/Write Byte Generator generates the read/write byte which is used in making a memory access. This byte is located at the leftmost byte position in the first data word sent to a core unit when a memory access is initiated. The remaining 3 data bytes consist of the core address.

The TP generates a memory access by constructing a core address in the AR, right justified. The read/write byte generator, when enabled, gates the necessary information into the leftmost byte of the AR, thus allowing the complete AR to be used as the source of the first data word during a memory access.

The read/write byte is all zeros if a memory read operation is to be performed. However, if the operation is a write, the bytes of the accessed double word which are to be written into, are represented by 1's in their respective bit positions in the Read-Write Byte. The placement of these 1's will depend on the size of the cell being written and its byte address. During the rest of this section, it should be remembered that the generation of the read/write byte is trivial for a read access (i.e. all 0's). All of the logic described is concerned only with the case of a write access.

The logic equations for the generation of the read/write bits (AR1 to AR8) are given below. They are derived from the table in Figure 4.2.2.6.



Cell Size				
Byte Address	CSB	CSH	CSW	CSD
000	1	1,2	1,2,3,4	1,2,3,4,5,6,7,8
001	2	CAE	CAE	CAE
010	3	3,4	CAE	CAE
011	4	CAE	CAE	CAE
100	5	5,6	5,6,7,8	CAE
101	6	CAE	CAE	CAE
110	7	7,8	CAE	CAE
111	8	CAE	CAE	CAE

CAE indicates cell alignment error

Figure 4.2.2.6 - Read-Write Bit Generating Table for the First Access  
(Shows which bits must be turned on for a given set of conditions)

$$\begin{aligned}
AR1 &= CSB \cdot 000 \vee CSH \cdot 000 \vee CSW \cdot 000 \vee CSD \cdot 000 \\
AR2 &= CSB \cdot 001 \vee CSH \cdot 000 \vee CSW \cdot 000 \vee CSD \cdot 000 \\
AR3 &= CSB \cdot 010 \vee CSH \cdot 010 \vee CSW \cdot 000 \vee CSD \cdot 000 \\
AR4 &= CSB \cdot 011 \vee CSH \cdot 010 \vee CSW \cdot 000 \vee CSD \cdot 000 \\
AR5 &= CSB \cdot 100 \vee CSH \cdot 100 \vee CSW \cdot 100 \vee CSD \cdot 000 \\
AR6 &= CSB \cdot 101 \vee CSH \cdot 100 \vee CSW \cdot 100 \vee CSD \cdot 000 \\
AR7 &= CSB \cdot 110 \vee CSH \cdot 110 \vee CSW \cdot 100 \vee CSD \cdot 000 \\
AR8 &= CSB \cdot 111 \vee CSH \cdot 110 \vee CSW \cdot 100 \vee CSD \cdot 000
\end{aligned}$$

where the three bit combinations indicate the last three bits of the address of the data being accessed. These equations are still highly redundant, since by the time they are used, the cell alignment checking logic will have rejected accesses involving misaligned cells. Thus, using this knowledge we can simplify these equations to the following form:

$$\begin{aligned}
AR1 &= 000 \\
AR2 &= (CSD \vee (CSW \cdot \overline{AR33})) \vee CSH \cdot 000 \vee CSB \cdot 001 \\
AR3 &= (CSD \vee (CSW \cdot \overline{AR33})) \vee CSBH \cdot 010 \\
AR4 &= (CSD \vee (CSW \cdot \overline{AR33})) \vee CSH \cdot 010 \vee CSB \cdot 011 \\
AR5 &= CSD \vee \overline{CSD} \cdot 100 \\
AR6 &= (CSD \vee (CSW \cdot \overline{AR33})) \vee CSH \cdot 100 \vee CSB \cdot 101 \\
AR7 &= (CSD \vee (CSW \cdot \overline{AR33})) \vee CSBH \cdot 110 \\
AR8 &= (CSD \vee (CSW \cdot \overline{AR33})) \vee CSH \cdot 110 \vee CSB \cdot 111
\end{aligned}$$

where  $CSBH = CSB \vee CSH$ .

These simplified equations have been implemented using the IC logic shown in Drawing 36-6 of the TP Logic Book.

#### 4.2.3 Memory Access Sequence

The purpose of this section is to explain the Memory Access sequence in terms of the operational and control point flowcharts. These descriptions incorporate the logic and techniques which were described in the previous section. The operational flowchart is described first. Then the control logic and control point flowcharts for each subsection of the Memory Access Sequence are explained.

#### 4.2.3.1 Memory Access Sequence Description

There are two possible entry points to the Memory Sequence. If the access is initiated through direct entry a check must be made to see if the immediate option is in effect. This option allows the pointer registers themselves to be operands. Thus if the immediate option is specified the data will be read into or out of the PR's or PRSNR's instead of core. The option is indicated by a fifth bit in the tag register. This bit is set to 1 when the tag register is loaded from an operand phrase which specifies an immediate operand.

The immediate operand sequence is quite straightforward. In the case of a memory read the proper Pointer Register field or fields are read out and gated into the LR, left justified. In a memory write the data is taken from the LR, in a left justified position, and gated into the proper field or fields of the Pointer Register selected by the tag register, TGR.

If the immediate option is not used, the first step of the direct entry sequence is to load the DR, right justified, with the pointer value of the requested PR and to set the PR segment name register selection logic to the proper PR segment name register.

The second entry of the memory access sequence enters at this point. Thus if the modified entry is used, the DR must contain the desired virtual address within the segment, right justified, and the PR segment name register selection logic must already be set to the proper PR segment name register before the sequence is entered.

At this point the DR (i.e. the virtual address) is stored in PR Segment Name Register 15, which is not used to hold a segment name since there is no PR#15. This address will be used later on in the memory sequence if an access must be made to the Segment Name Table or if the data being accessed is in partitioned mode. It is also used during memory sequence interrupts. While the DR is being stored in PR Segment Name Register 15, the cell alignment check is also performed and if an error is detected an interrupt is initiated.

Next the base register association logic is activated and eventually gives a reply stating whether or not the base information for the desired

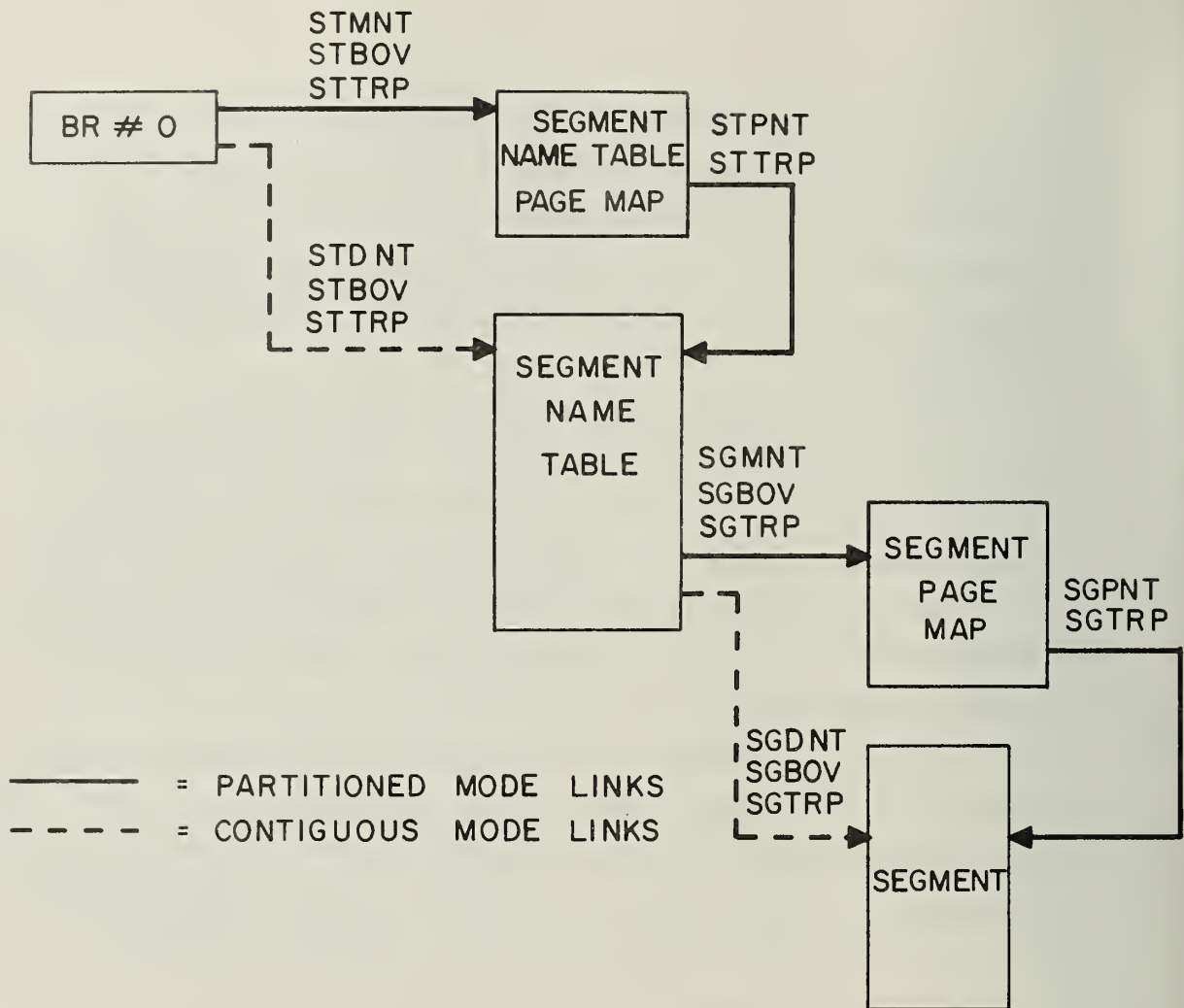
segment is present in one of the hardware registers. If it is not, a memory access must be made to the Segment Name Table to get the proper base information. The segment name itself is used as an address to access the Segment Name Table. It should be noted that all segment names should end in 00 in the rightmost bits since it is desired to make accesses on word boundaries.<sup>1</sup>

During the Segment Name Table access the standard checks are made on the access mode and the bounds. If one of these checks is violated, the Segment Name Table version of the interrupt is set and the Memory Interrupt Sequence is started. The interrupts which are possible at this point are STBOV, STTRP, and either STMNT or STDNT depending on whether the Segment Name Table is in Partitioned Mode or not. The occurrences of these interrupts are shown in Figure 4.2.3.1/1.

If the Segment Name Table is in partitioned mode, the address in the DR, i.e. the segment name, must be converted using the page table and the Partitioned Mode Address Conversion sequence (see below). During this sequence, if the access mode is trap or not there, then the proper Segment Name Table interrupt is executed, (i.e. STTRP or STPNT).

---

<sup>1</sup>It has been suggested that each segment name be shifted left 2 bits but this cannot be done because the LR would have to be used. During the first part of the memory sequence the LR contents must not be destroyed because the calling sequence sometimes stores data there. One could save the LR in the AR before the shifting and then restoring it afterwards but this tactic involves additional time and logic. The present restriction upon segment number encoding seems much simpler.



STMNT - SEGMENT NAME TABLE PAGE MAP NOT THERE  
 STDNT - SEGMENT NAME TABLE DATA NOT THERE  
 STBOV - SEGMENT NAME TABLE BOUNDS OVERFLOW  
 STPNT - SEGMENT NAME TABLE PAGE NOT THERE  
 STTRP - SEGMENT NAME TABLE TRAP  
 SGMNT - SEGMENT PAGE MAP NOT THERE  
 SGDNT - SEGMENT DATA NOT THERE  
 SGBOV - SEGMENT BOUNDS OVERFLOW  
 SGPNT - SEGMENT PAGE NOT THERE  
 SGTRP - SEGMENT TRAP

Figure 4.2.3.1/1 - Memory Sequence Addressing Interrupts



Eventually if there have been no interrupts, (or the interrupts have been processed) the requested base descriptor (one word) will come back from the memory and be stored in the base register whose queue counter is six. This is always the base register which has been inactive for the longest length of time. The associative register connected with this base register is loaded with the segment name, the DR is restored from PR segment name register 15, and the associative logic is reactivated. This time of course there will be a match.

Once the match has been obtained, either with or without recourse to accessing the Segment Name Table, the BR which matches is read out on to the DB, and the DB and DR are added together to complete the actual core address desired. Simultaneously with this the bounds is compared with the page address in the DR (see Section 4.2.2.3) and the queue counter logic is activated to recalculate the counter settings in view of this access (see Section 2.5.1.2). The Queue Counter Sequence starts by resetting the cycle counter. Then it repeatedly counts up on all of the queue counters until the counter which is assigned to the base register just selected overflows. At this point all the counters in which an overflow has occurred are counted up once. Then all the queue counters are again repeatedly incremented until each counter has overflowed (i.e. 8 counts all together). At this point the overflow bits are all reset, the selected counter is reset and the sequence is complete.

After the addition and bounds check have been completed, all the interrupt conditions have to be checked, and then a check is made for partitioned mode. If the segment is in partitioned mode the Partitioned Mode Address Conversion sequence must be used. Note that in this case the sum which was just calculated is of no use since the addition was performed under the assumption than the segment was in contiguous pages in core. However the bounds check which was computed in parallel serves to establish the validity of the virtual address.

In the partitioned mode address conversion sequence, the virtual address, which is in the DR, is right shifted one byte and then left shifted one bit while inhibiting the rightmost bit. This creates in the DR an address which points to the n'th halfword in core, where n is the virtual page referenced in the virtual address. During this shifting operation the LR must be used. However since it is desired to save the contents of the LR during the memory sequence it must be saved in the AR while the shifting is going on.

After the LR has been restored the address of the halfword entry in the page map is added to the chosen base register to give the physical core address of the halfword entry. Then an access to the memory is made.

When the data returns it is stored, left justified, in the DR. Then the contents of the LR is again temporarily saved in the AR.

The next step is to load the LR with the original virtual address in such a manner that the lower half of the address (i.e. the address within the physical page) is the third byte from the left (i.e. byte number 2). Then the DR is merged into the LR to form a 3 byte absolute core address, left justified.

Next the DR is loaded with the contents of the AR, i.e. the original LR contents at the beginning of the sequence. Then all that remains is to gate the address in the LR into the AR while permuting it so that it is right justified, and to gate the DR to the LR. Then the sequence can return with the final desired absolute core address in the AR, right justified. The actual memory core access can begin as soon as the Read/Write byte and the Exchange Net control byte have been set.

The detailed sequencing for the read cycle is as follows:



- R1. A request from the TP is sent to core via the Exchange Net in parallel with the access control byte and data address.
- R2. The reply from the Exchange Net will originate in the core unit and will signify:
- a) the core unit is connected to the TP;
  - b) the address and access control bytes have been received by the core unit and are held in a core unit register.
- R3. On receiving the reply the TP:
- a) removes the address etc., from the INBUS to the Exchange Net;
  - b) waits for the transmission of the required words from the core unit.
- R4. At this point the exact sequencing depends on the cell size of the cell being accessed:
- Byte, Halfword, or Word: only one word is received. It is gated to the LR, right justified, and the sequence is done.
- Double Word: two words are received. The first is gated to the LR, the second to the DR.
- R5. The instruction being executed may now operate on or dispose of the required cells in whatever manner necessary.

The write cycle is similar to the read cycle as far as addressing is concerned. The detailed sequencing for the write cycle will occur as follows:

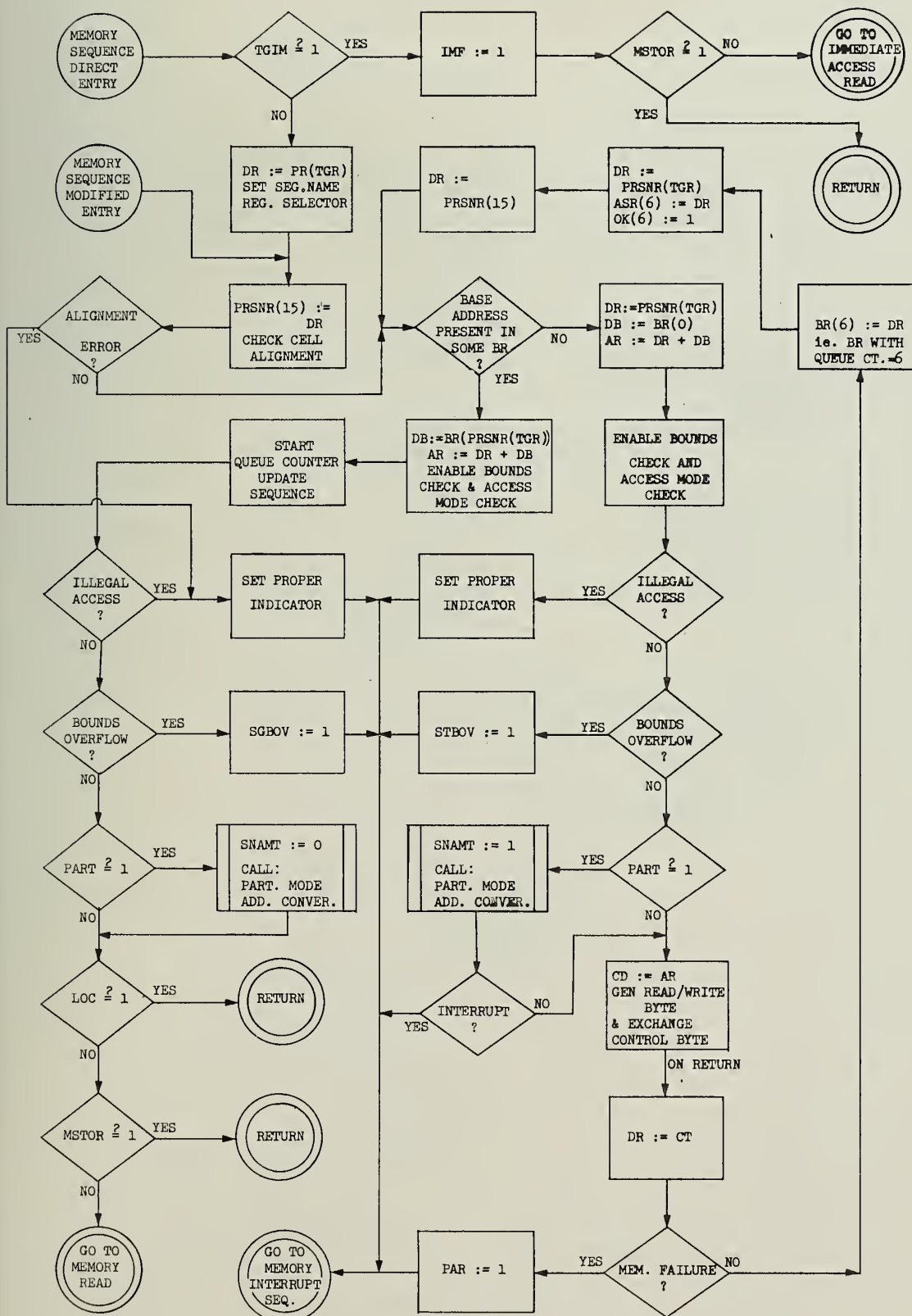
- W1. After the address has been constructed and bounds checked, a "load" registers signal is generated which causes the LR (and DR if the cell is a double word) to be loaded with the word(s) to be written by whatever sequence called the write-memory access sequence.
- W2. A request from the TP is then sent to core via the Exchange Net in parallel with the access control byte and data address.
- W3. The reply from the Exchange Net will originate in the core unit and will signify:
- a) the core unit is connected to the TP.
  - b) the address and access control bytes have been received by and are held in a core unit register.
- W4. Upon receiving the reply, the TP:
- a) removes the address, etc., from the INBUS
  - b) immediately gates out the first word to be stored.
- W5. When a "received" reply comes back from the core unit, the TP places on the INBUS the second word to be stored, if this is needed.
- W6. On receiving the second reply, or if a second transmission was not necessary, the first reply, the TP disconnects.

It should be noted that the number and timing of the word accesses will, as in the read sequencing, depend on the cell size and starting byte address of the cell being processed. When a word of memory is stored, only those bytes indicated in the Read-Write byte are actually written into core.

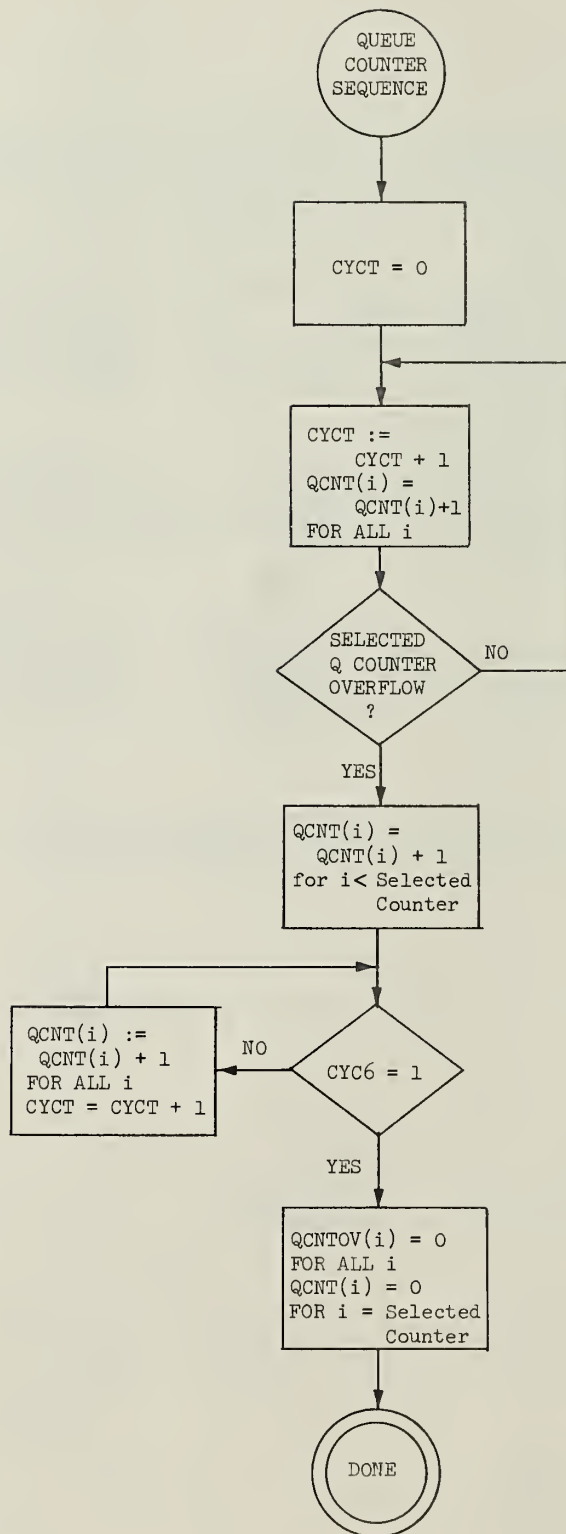
In the memory sequence all the interrupts except those dealing with memory unit errors are detected before the memory unit is ever accessed and therefore nothing in core is changed. Also since the PR's are only changed in immediate operand situations where there can be no interrupts, no part of the data base will have changed. Thus if an interrupt is detected the whole sequence can be restarted. Figure 4.2.3.1/1 shows the possible memory organizations and the points at which various interrupts might occur. A list of the other possible interrupts is given in Figure 4.2.3.1/2. Note that any memory unit interrupt (i.e. parity error, etc.) is considered catastrophic and no attempt will be made to save memory sequence information which would be needed to continue the instruction on from the point of interrupt.

<u>Interrupt Code</u>	<u>Meaning</u>
ILAC	Illegal access - attempt to write in read only storage.
PAR	Parity Error.

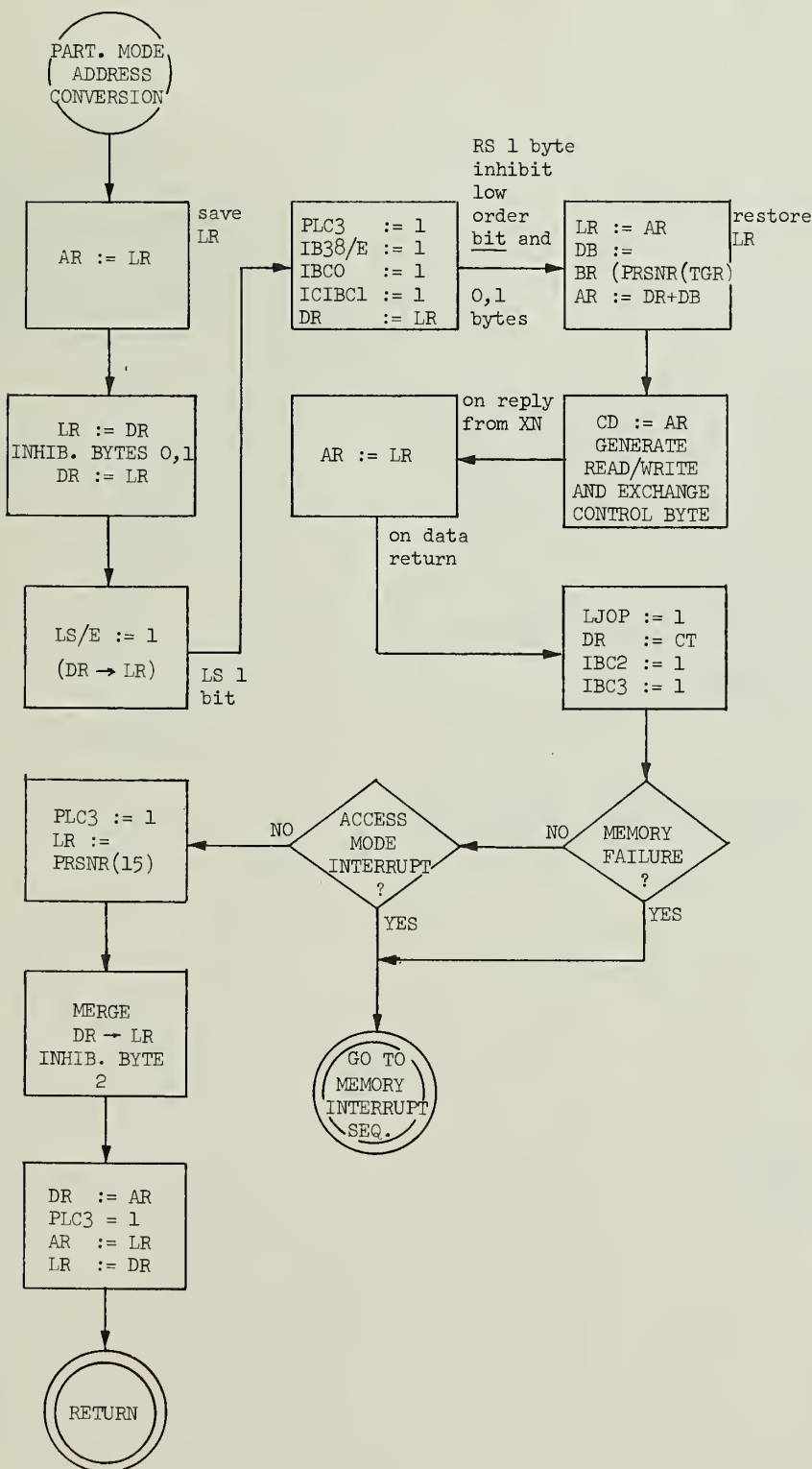
Figure 4.2.3.1/2 - Memory Interrupts Other Than  
Addressing Interrupts



Memory Sequence - Initial Address Construction

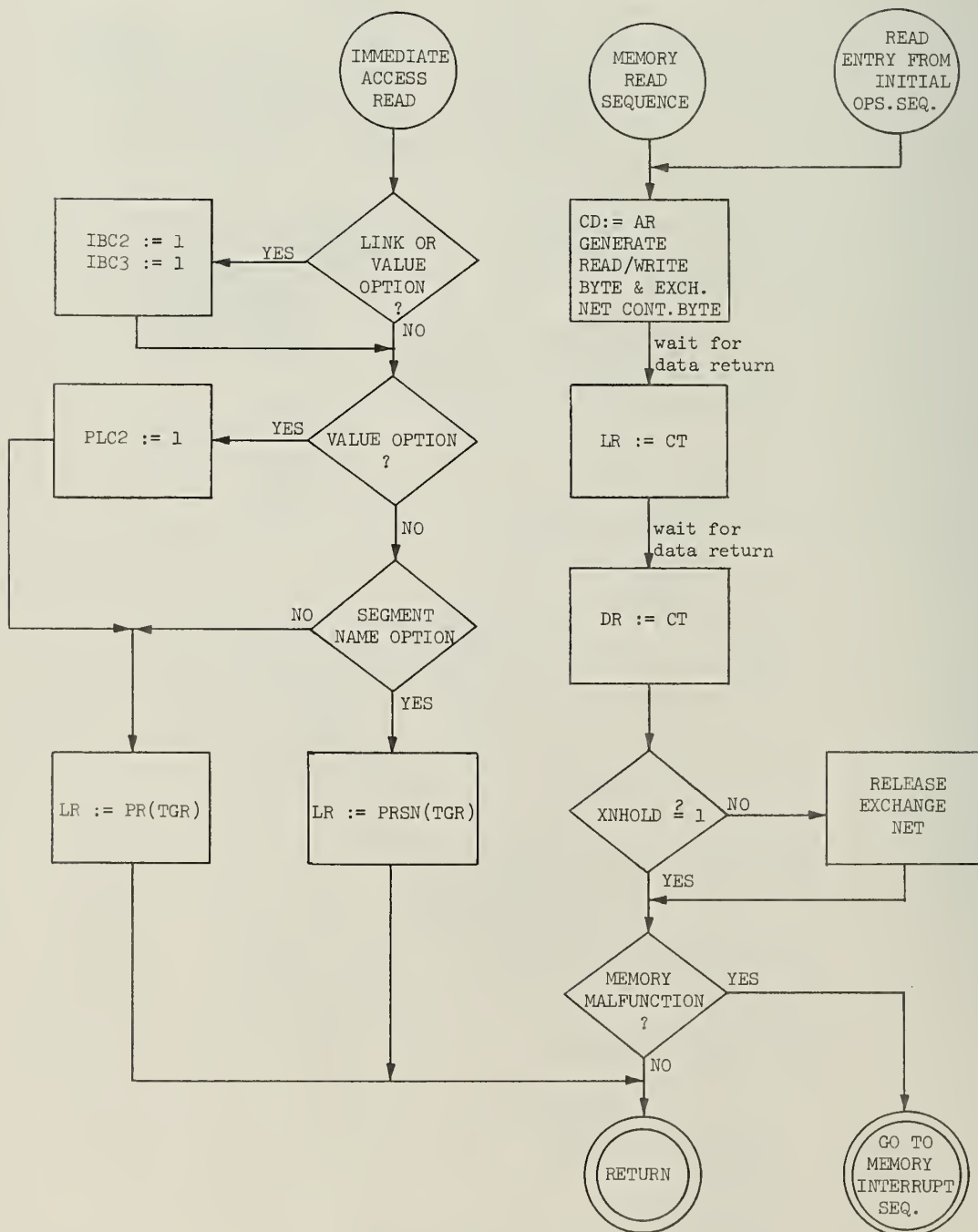


Memory Sequence - Queue Counter Update Sequence

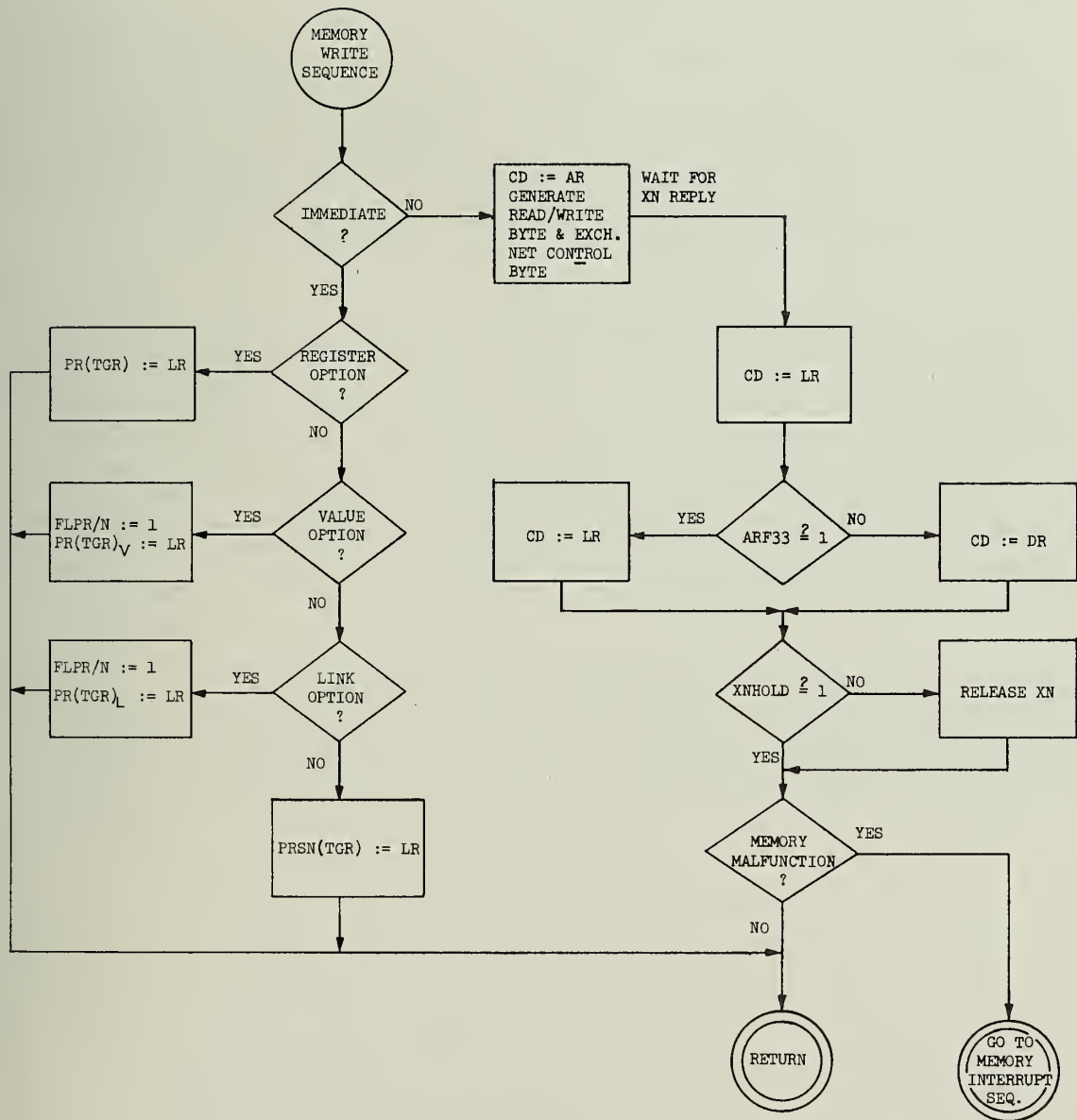


Memory Sequence - Partitioned Mode Address Conversion









Memory Sequence - Write

#### 4.2.3.2 INITIAL ADDRESS CONSTRUCTION Control Logic

The Initial Operations sequence performs the initial address construction in the memory sequence and checks for the various possible non-hardware interrupts. Referring to logic drawings 36-1a through 36-3 the only non-straightforward part of the logic is the MT3-MT4-MT5 complex which is used to check for the presence of the desired base information, and the MT10-MT11 complex which accesses the exchange net.

MT3 is actually a normal control point except for the fact that the output of the delay circuit is not used to reset its flip-flop. Instead the Advance Out,  $\overline{A}_O$ , is fed to one of two other delay circuits depending on whether or not the desired base information was present. At the same time one of two task signals is turned on to perform the desired operations. The outputs from the two delay circuits are or'ed together and fed back to be AND'ed with the GODELY signal and eventually to reset the flip-flop in control point MT3.

Thus the overall affect is to first activate the association logic for the base registers using MT3. Then when this causes the DA and NYET signals to be valid the choice between control points MT4 and MT5 can be made. Note that in either case it is still necessary for the task signals turned on by MT3 to remain on while MT4 or MT5 is activated.

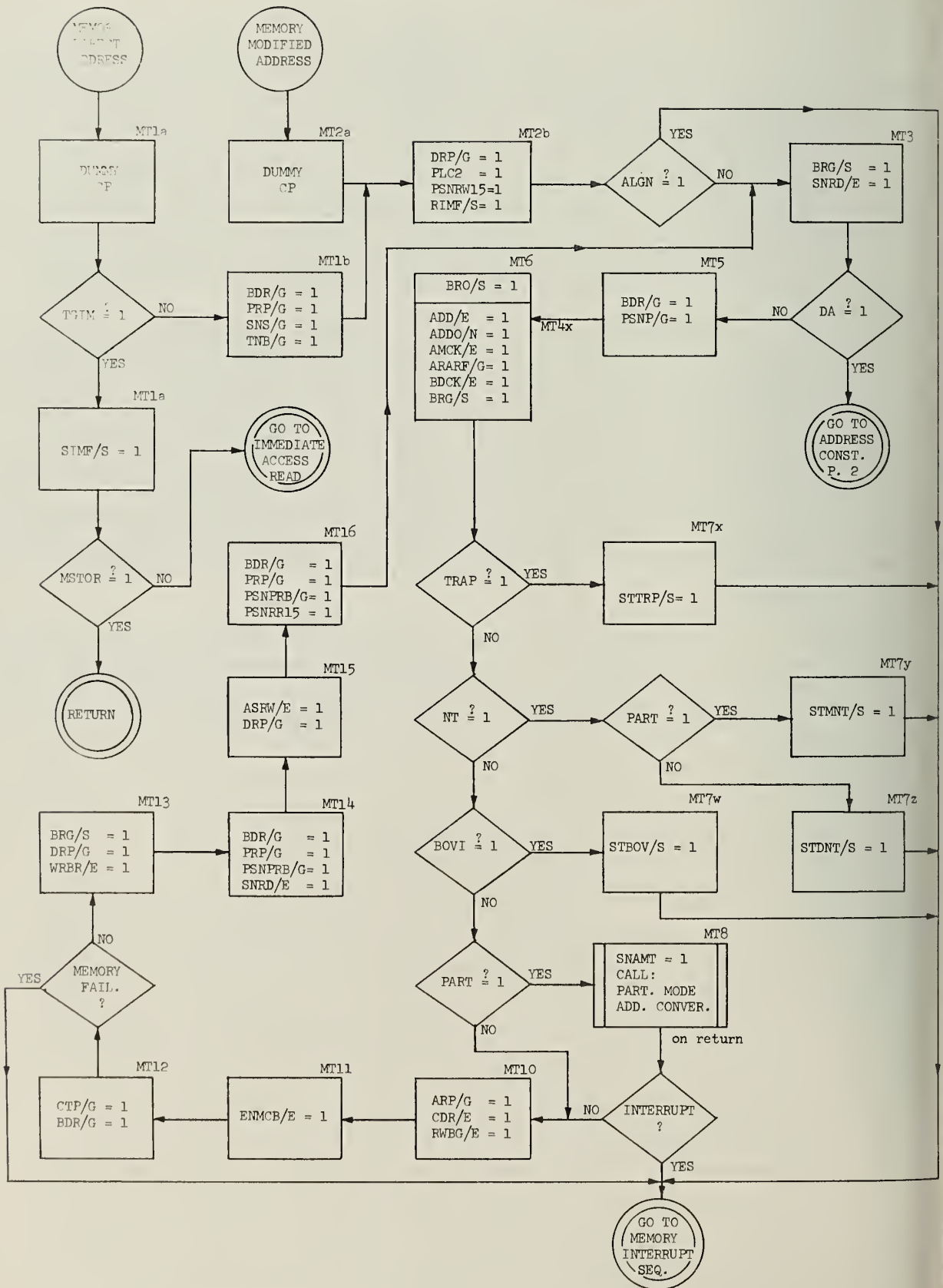
The MT10-MT11 complex might be called a hybrid calling control point. It is important to understand its operation since this type of logic will be used whenever the Exchange Net is accessed in order to get to core memory. The complex is a calling control point in that it makes use of normal return and interrupt return signals. However, it also acts like a regular control point in that it uses a delay. To make matters even more confusing there are two task signals.

The first task signal, MT10, as shown in Figure 4.2.3.2, is used to gate the AR to the cable drivers and activate the read/write byte generator. The delay circuit then waits a suitable length of time until the read/write byte is also present on the cable driver lines. Later on it may be possible to eliminate this delay if it turns out that read/write byte generation can be done in parallel with the exchange net request.

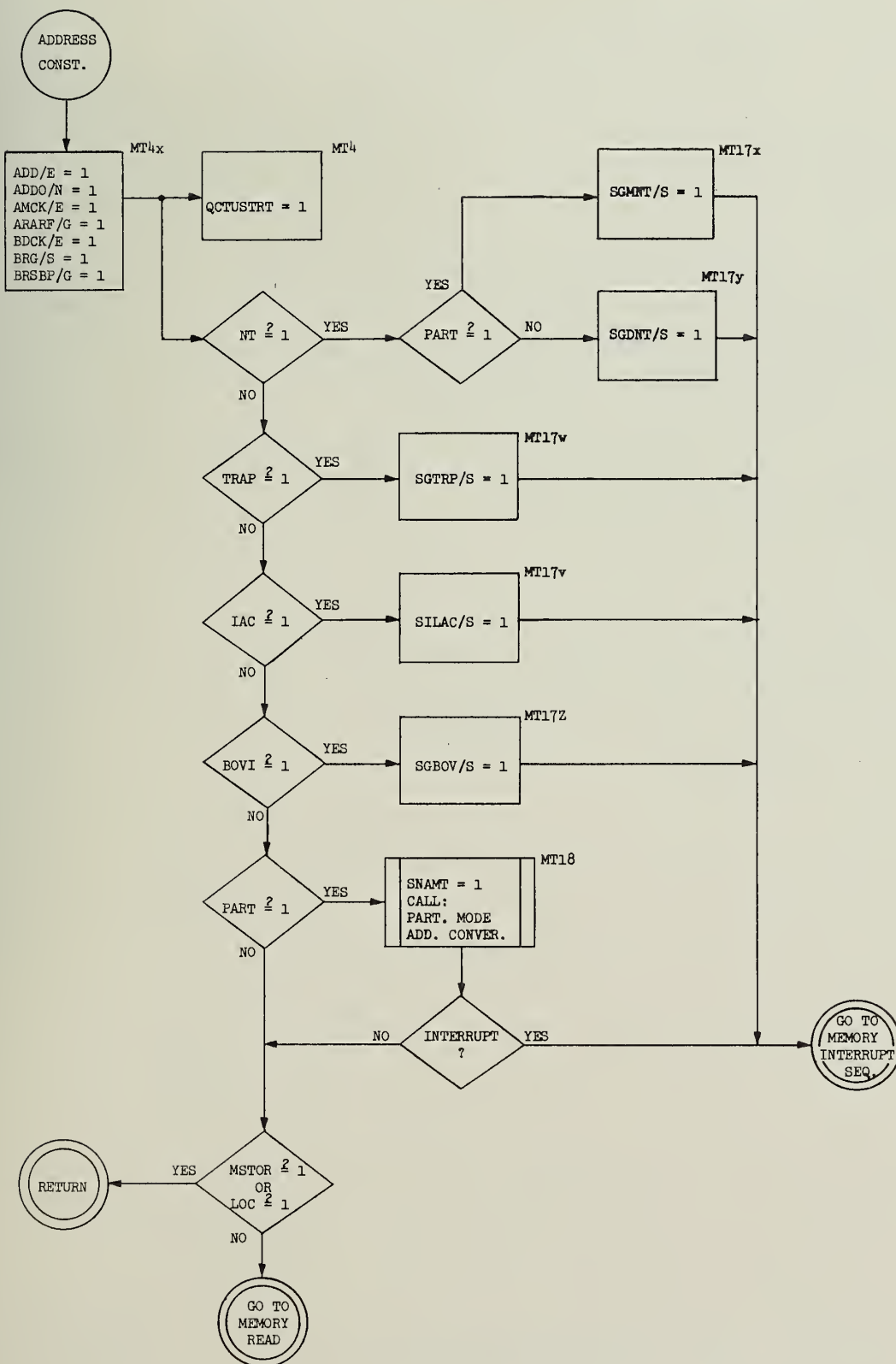
The second task, which begins after the delay has finished, generates the Exchange Net control byte and makes a request for the proper memory box. The control point then remains active until the return signal is activated. In this case the return signal will be the signal that the memory cycle has started. This signal will reset the two control task signals and either activate the next control point if no data error occurred, or generate an interrupt if there was one.

If no reply is received from the exchange net, a timer (see logic drawing 37-2) will eventually run out and generate an Exchange Net No Reply signal, XNNRP, which is sent to the interrupt return input to the calling control point. This will cause the control point to reset and turn off its task signals. At the same time, the no reply signal is also used to automatically activate the Memory Interrupt Sequence. Thus the  $\overline{I}_O$  output of the Calling Control point need not be connected.

This same technique is used if the memory or any other external unit does not reply to a TP request within some predetermined length of time (usually quite long relative to a single access of the particular unit).



Memory Sequence - Initial Operations - Control Step Flow Chart

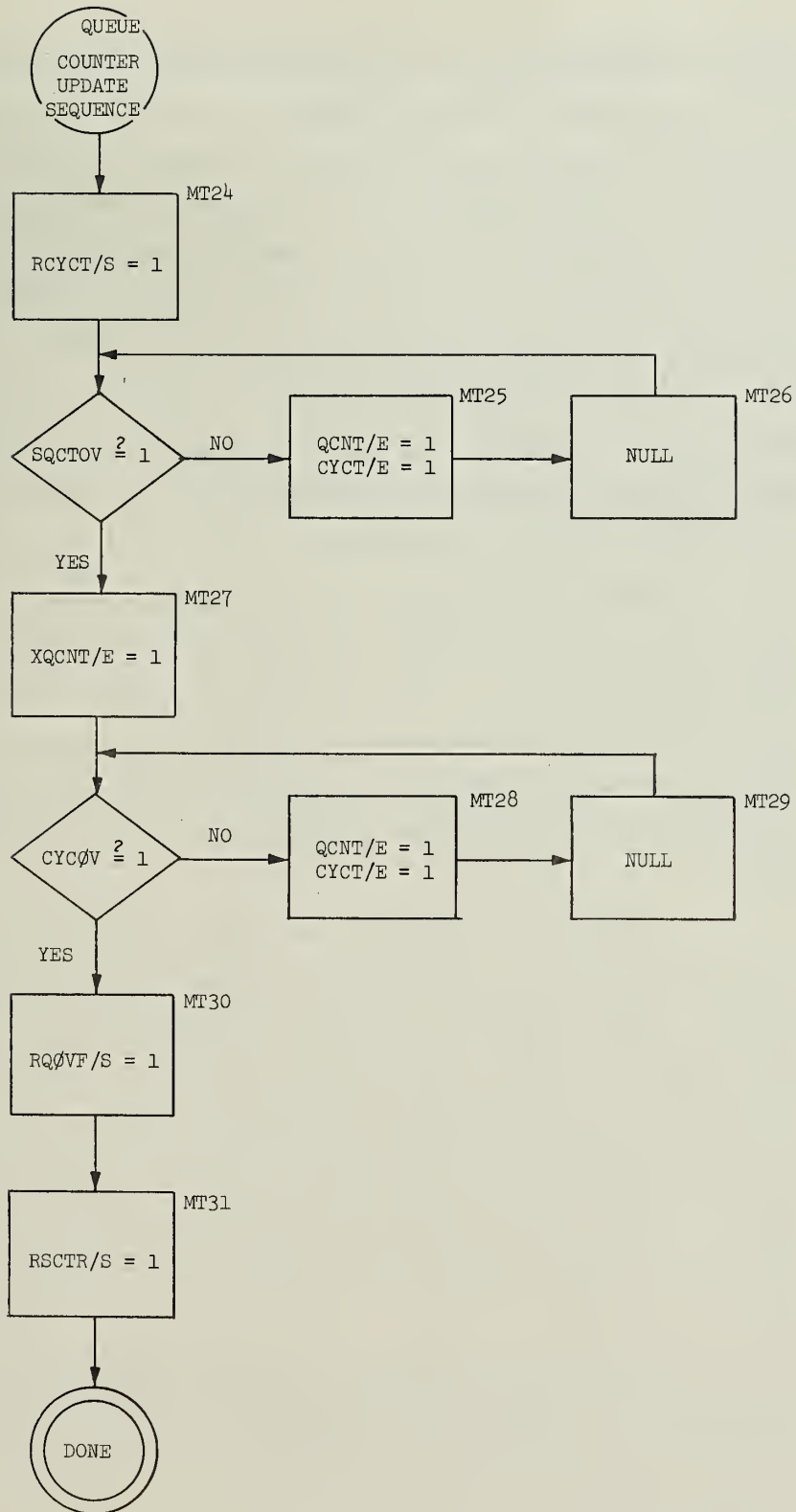


#### 4.2.3.3 QUEUE COUNTER UPDATE Control Logic

The Queue Counter Update sequence is used to update the status of the queue counters for the hardware base registers in the TP each time one of these registers is used. It is a fairly simple sequence. Two "null" control points are used (MT26 and MT29) in the counting loops.

Since the control sequence is designed to work independently of all the other control logic once the sequence has been activated, there is no return signal generated at the end of the sequence. There is also no need to use a calling control point to start the sequence.

It should be noted that if this sequence is activated with a task signal, the actual operation will not start until the task signal returns to its "steady state" of "1".



Memory Sequence - Queue Counter Update Sequence  
Control Step Flow Chart



#### 4.2.3.4 PARTITIONED MODE ADDRESS CONVERSION Control Logic

The Partitioned Mode Address Conversion Sequence is used to access the page map in a Partitioned Mode access and construct the actual physical core address which will be needed to access core. At the same time it checks the access bits connected with the page map entry to be sure the access is a valid one. The sequence utilizes one control signal, SNAMT, and one internal control flip-flop, PMAREP.

SNAMT, the Segment Name Table signal, is used to indicate that the sequence is being used to access a Segment Name Table entry. This information is important in determining which interrupt indicator to set if an interrupt condition occurs.

PMAREP, the Partitioned Mode access repeat flip-flop, is used to control the operation of the page map accessing loop. In the case of a virtual page overflow access in Partitioned Mode, the page map must be accessed twice, once for each page on which the data appears. During the second access PMAREP is set to one so that the flip-flop can be used to control several different operations in the second access of the page map.

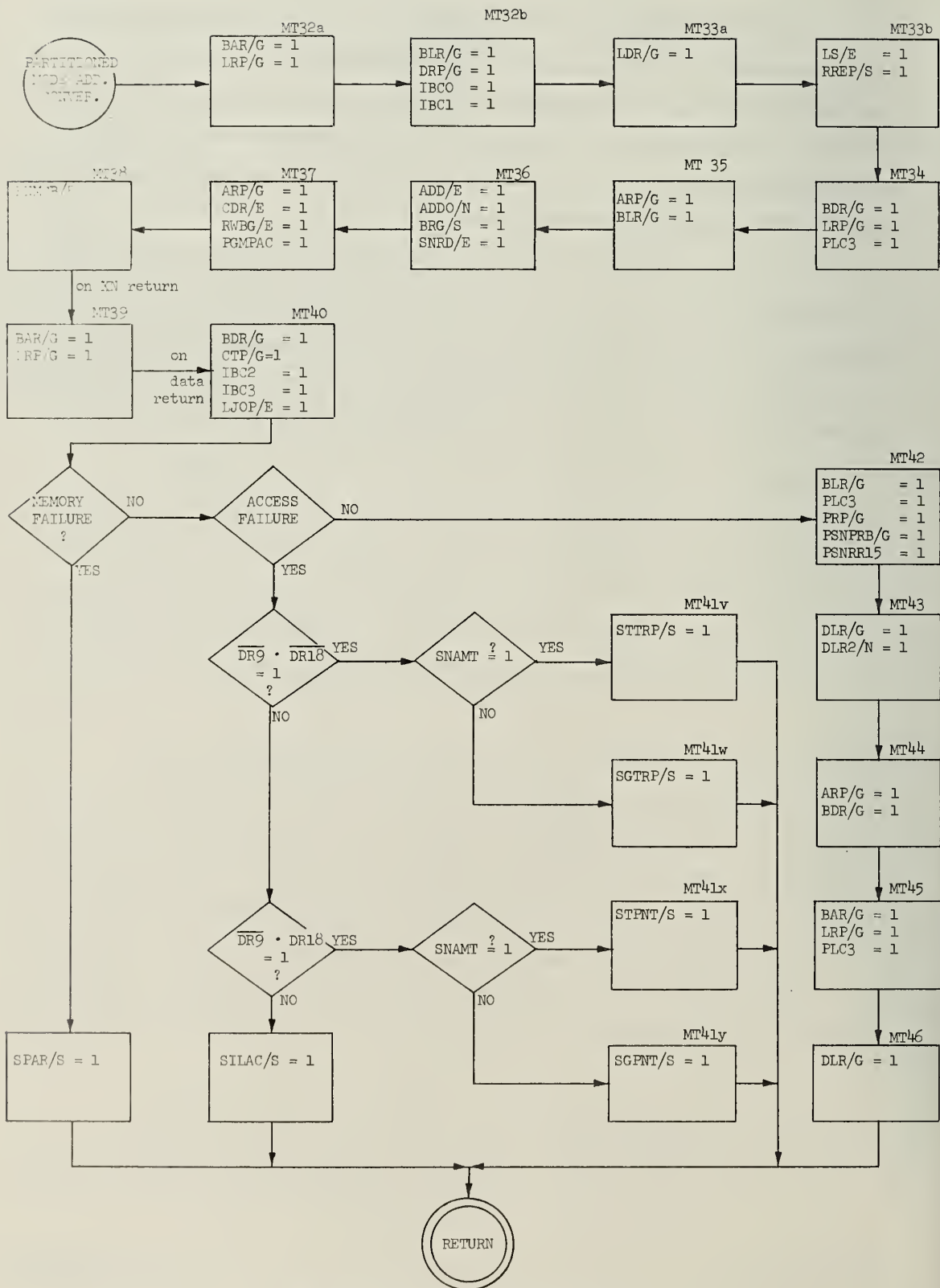
The MT37-MT38-MT39 control point complex is another example of the use of complex control points to generate the Exchange Net control signals. MT37 generates the proper read/write byte and then MT38 makes the request for service while MT37 is still on. After a response is obtained from the Exchange Net, MT39 will cause the LR to be gated to the AR unless the VPGOV flip-flop is on and it is the first access to the page map. When this is done, the control waits for the return signal from the core box indicating that the data is on the data lines waiting to be read.

In MT40 the delay must be set long enough for the access mode checking logic to produce valid outputs so that the decision logic in front of MT41 will give the correct results.



Control Point MT<sup>4</sup>1 is used to set any interrupt indicators if this becomes necessary. The logic in front of it is used to test the access mode of the page map entry which is, at that time, stored in the DR, left justified. If any interrupt condition occurs, MT<sup>4</sup>1 is activated and the proper indicator is turned on according to what type of condition caused the interrupt.

Note that the interrupt signal for the memory sequence as a whole is turned on after MT<sup>4</sup>1 has set the proper indicator. This is done to save time and logic since there is no need to have a special Partitioned Access Mode sequence interrupt return when all it would do is activate the Memory Sequence interrupt return anyway.



Memory Sequence - Partitioned Mode Address  
Conversion Sequence Control Step Flow Chart

#### 4.2.3.5 Memory READ Sequence Control Logic

The Memory READ sequence is a somewhat unconventional sequence. In its most commonly used access it is not a sequence at all, but merely an extension of the Initial Operations Sequence. In this case there is no activation of a calling control point and the read operation will begin at either M48 or M50 depending on whether an immediate or normal access will be performed. There is one control signal for the READ access sequence, XNHOLD. If this signal is on the sequence will not release the Exchange net when it is finished.

It is sometimes necessary, however, to call the Memory READ sequence as an actual sequence. In this case the calling control point will cause the activation of dummy control point M49 which in turn activates M50. Note that immediate address is not possible on a direct call to the Memory READ sequence.

The purpose of the Immediate Address entry is to directly access the pointer register fields instead of making a core access if this option is specified by the operand phrase. In this case the TGIM signal will be "1". If MSTOR = 0, i.e. a memory read, the main memory sequence, Initial Operations, will detect that an immediate read is to be performed and will directly activate M48 of the READ sequence.

In an immediate read access there are actually four possible cases which may occur depending on the field designator bits (IR34 and IR35). Six NAND's are used to decode these IR bits. At any given time during the memory sequence only one output from the logic will be "0". In all four cases the PRP/G, BLR/G, and TNB/G signals must be set to "1" to enable the gate from the PR to the LR. In addition, if the field designator indicates a link variant (IR34 = 0, IR35 = 1), the PLC2 signal, which causes the data to be permuted left 2 bytes, must be set to "1". Finally if the field designator is not a register or segment name variant (i.e. it is either a link or value) the right most two bytes into the LR must be inhibited by setting IB2/E and IB3/E to "1".

Once the proper PR field(s) is loaded into the LR, the READ sequence activates the main memory sequence return signal.

Due to the nature of the timing requirements for the Exchange Net and the core memory units, the operation of the READ control logic for a conventional memory access is fairly complex, although the logic itself is fairly simple. The Exchange Net request made by complex control point M50-M51 is a typical example of the technique described in Section 4.2.3.2. Referring to Figure 4.2.3.5/1, the first task signal in this type of complex control point is used to gate the needed data and read/write byte to the cable drivers while the second task signal generates the control byte once this data is valid. The reply from the Exchange Net is used for the normal return while the Exchange Net No Reply signal is used for the interrupt return. Since the Exchange Net-TP logic takes care of the interrupts in these cases, the Interrupt out signal from the control point is not connected and is only used to reset the control point.

The technique used to obtain the data from the memory units is somewhat unconventional. Since the memory units always send first the left half and then the right half of the double word accessed during the memory cycle, it is necessary for the control sequence to know which word it wants at any given time.

The method used for obtaining data in a memory read is shown in Figure 4.2.3.5/2. After the Exchange Net has replied to the TP's processor request (complex control point M50-M51), the memory control sequence has about 300 ns. before it must accept the data coming back from the memory units. During this wait the M52 control point is activated. When this control point is first activated the data will not be ready. It is important in designing the control sequence to make sure that this control point is activated before the data has arrived from core since otherwise timing errors will result and incorrect data may be loaded into the TP register.

The MT52 task signal will turn on the necessary gates to transmit the data from the Exchange Net cable terminators into the LR. These signals will remain on until the return input of the control point is activated.

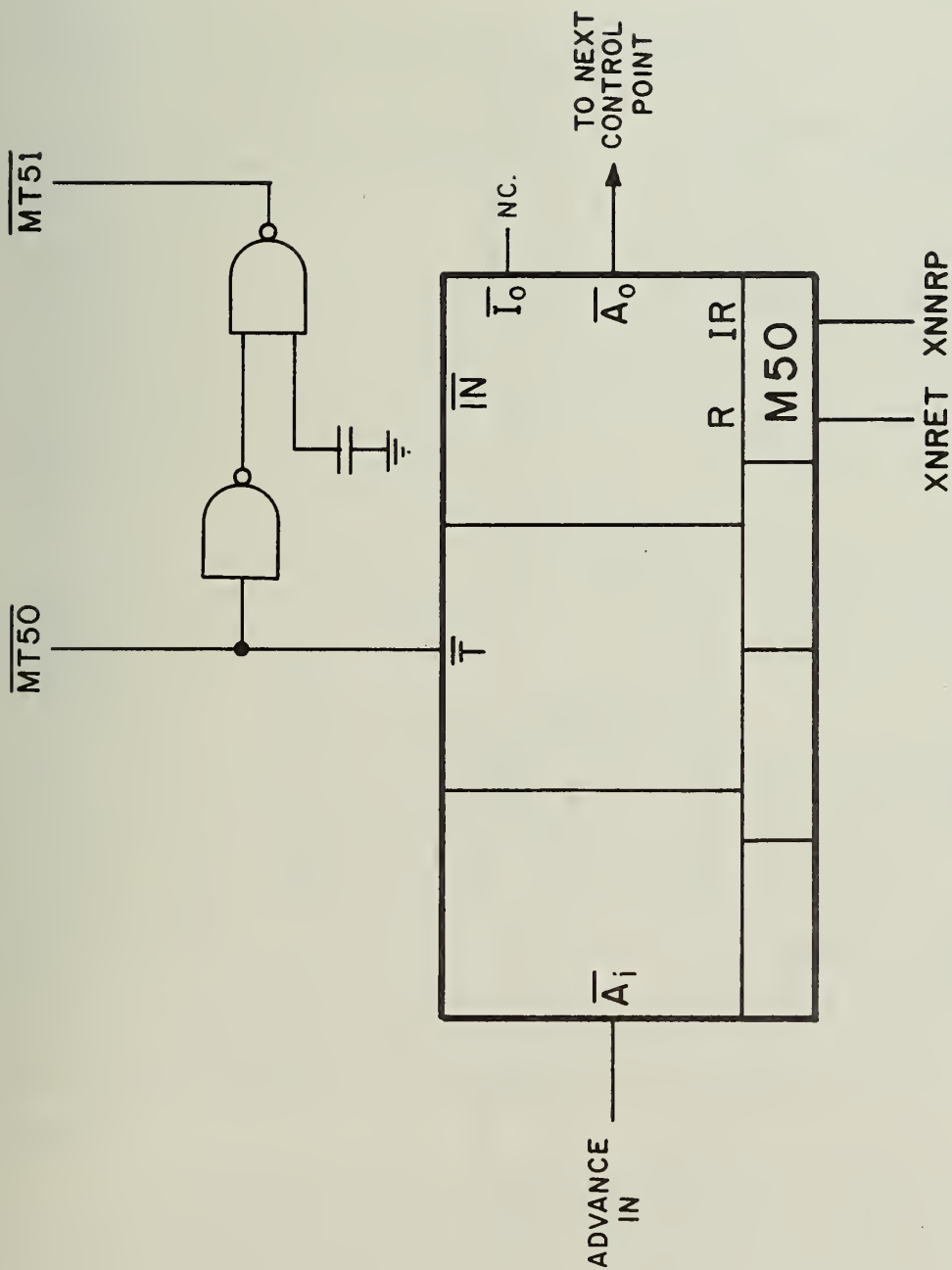


Figure 4.2.3.5/1 Typical Implementation of an Exchange Net Access

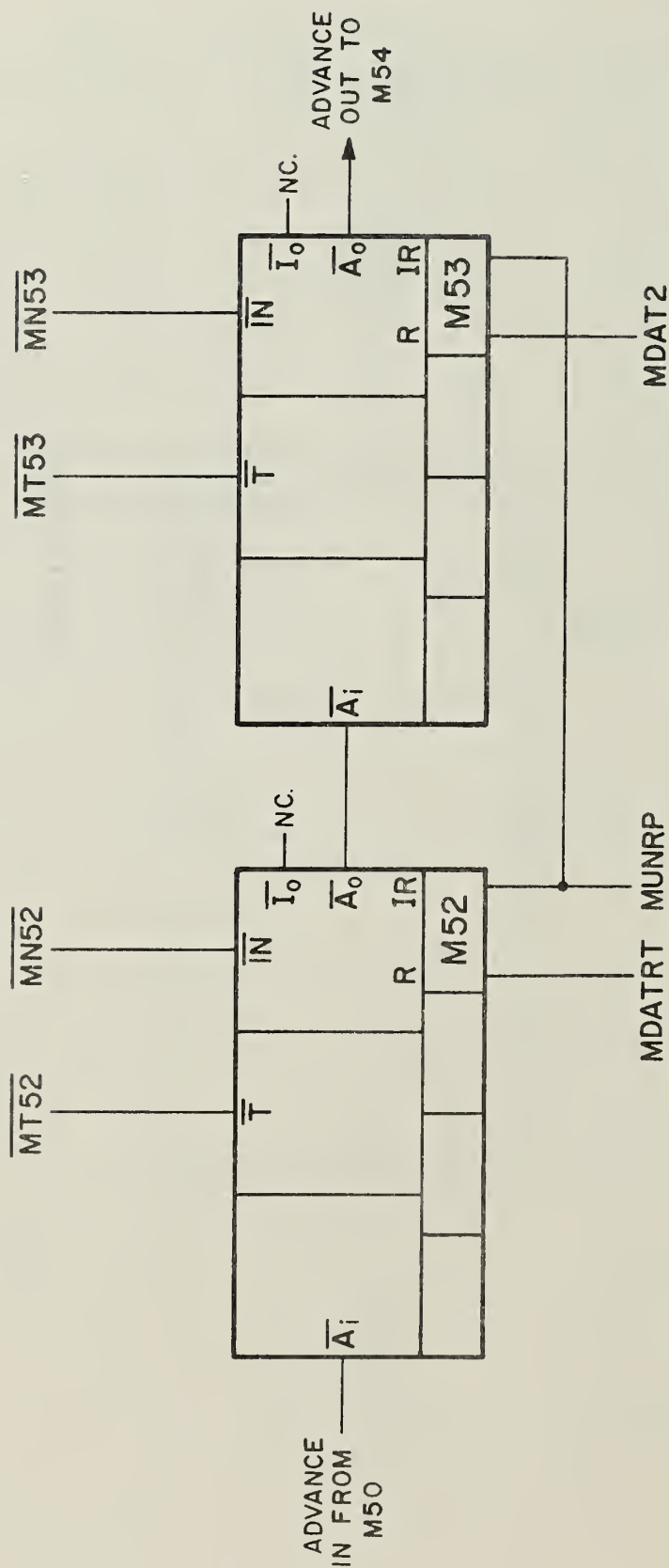


Figure 4.2.3.5/2 Implementation for Accepting Returning Information During a Memory Read Access



This is done using a signal which turns on a fixed amount of time after the TIO signal from the memory unit has indicated that the data lines are valid and contain the desired word from core. The fixed amount of time is adjusted so as to allow the data to travel through the TP permuter and into the desired register before the control point is turned off.

The return signal used to turn off the control point is generated by the TP-Exchange Net Interface Logic. This signal, MDATRT, waits for either the first or the second word to be returned from the memory unit depending on the alignment of the cell being accessed. This can be determined from the 3rd lowest order bit of the address which at this time is contained in ARF33. For example, if ARF33 is '0', the desired data will be in the first word returning from the memory unit. If ARF33 = '1', it will be in the second. ARF33, in effect, indicates in which half of the double word being accessed, the desired cell begins.

The logic necessary to generate the MDATRT signal is shown in figure 4.2.3.5/3. Note that MDAT1 and MDAT2 turn on a constant length of time after the first and second data words, respectively, have returned from core.

The second data word will be gated to the DR by M53. Note that this control point uses MDAT2 as a return signal and must have 2 inverters so that the delay time for it and for MDATRT are equivalent. If it turns out that ARF33 = '1' and MDATRT did not turn on until MDAT2 was activated, then MDAT2 will still be on at the time M53 is activated. This will cause M53 to turn off right away and garbage will more than likely be loaded into the DR. However this does not matter since it will only occur on non-double word accesses and in these cases the contents of the DR are not used anyway.

Note that if in M52 or M53 the unit never responds, the interrupt line will be turned on by the memory unit no reply signal. This is done to turn the control point off without activating the next one rather than to make an interrupt, since the interrupt itself will be generated by the Exchange Net-TP Interface logic. Thus the Interrupt Out signal is not connected.



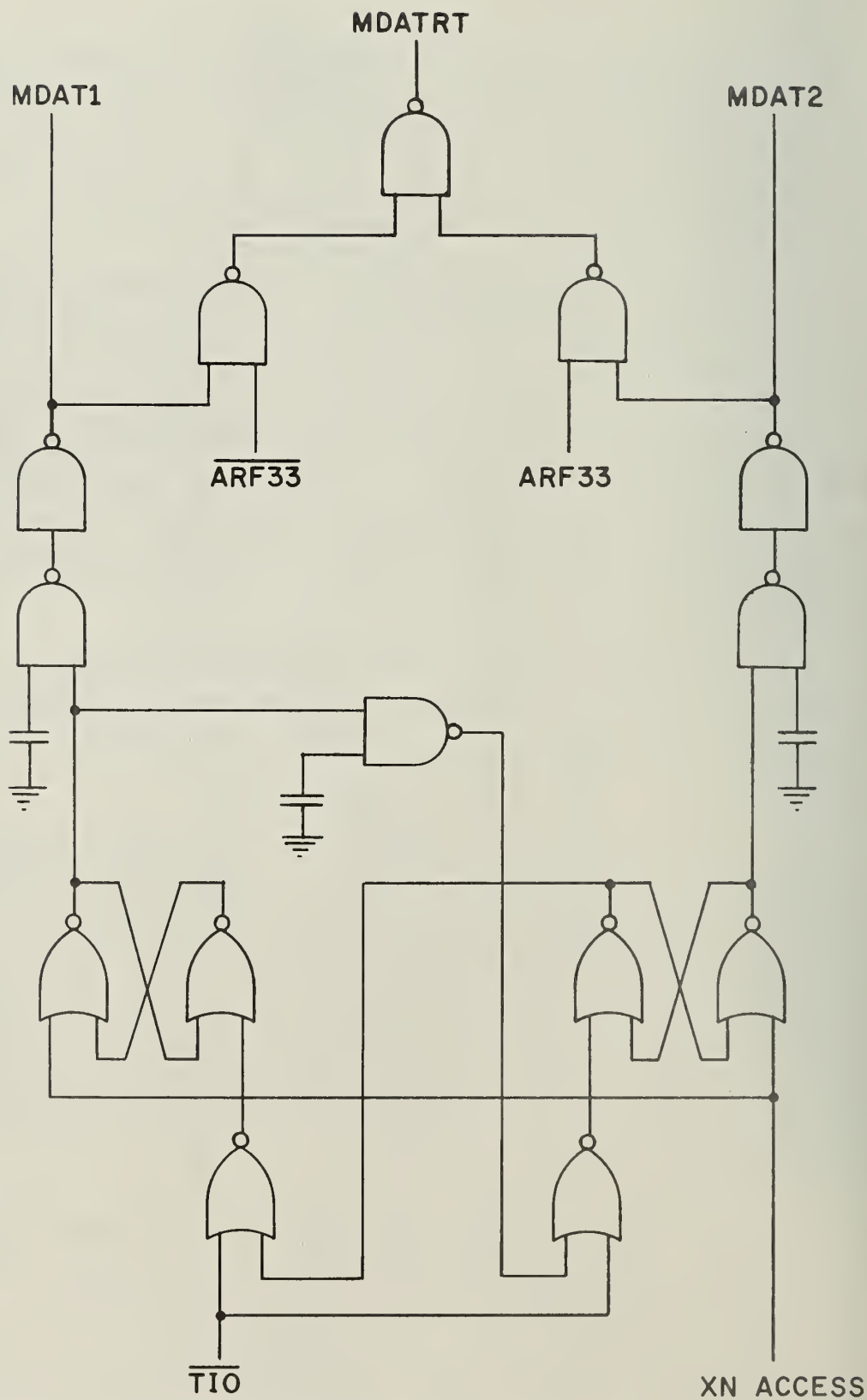
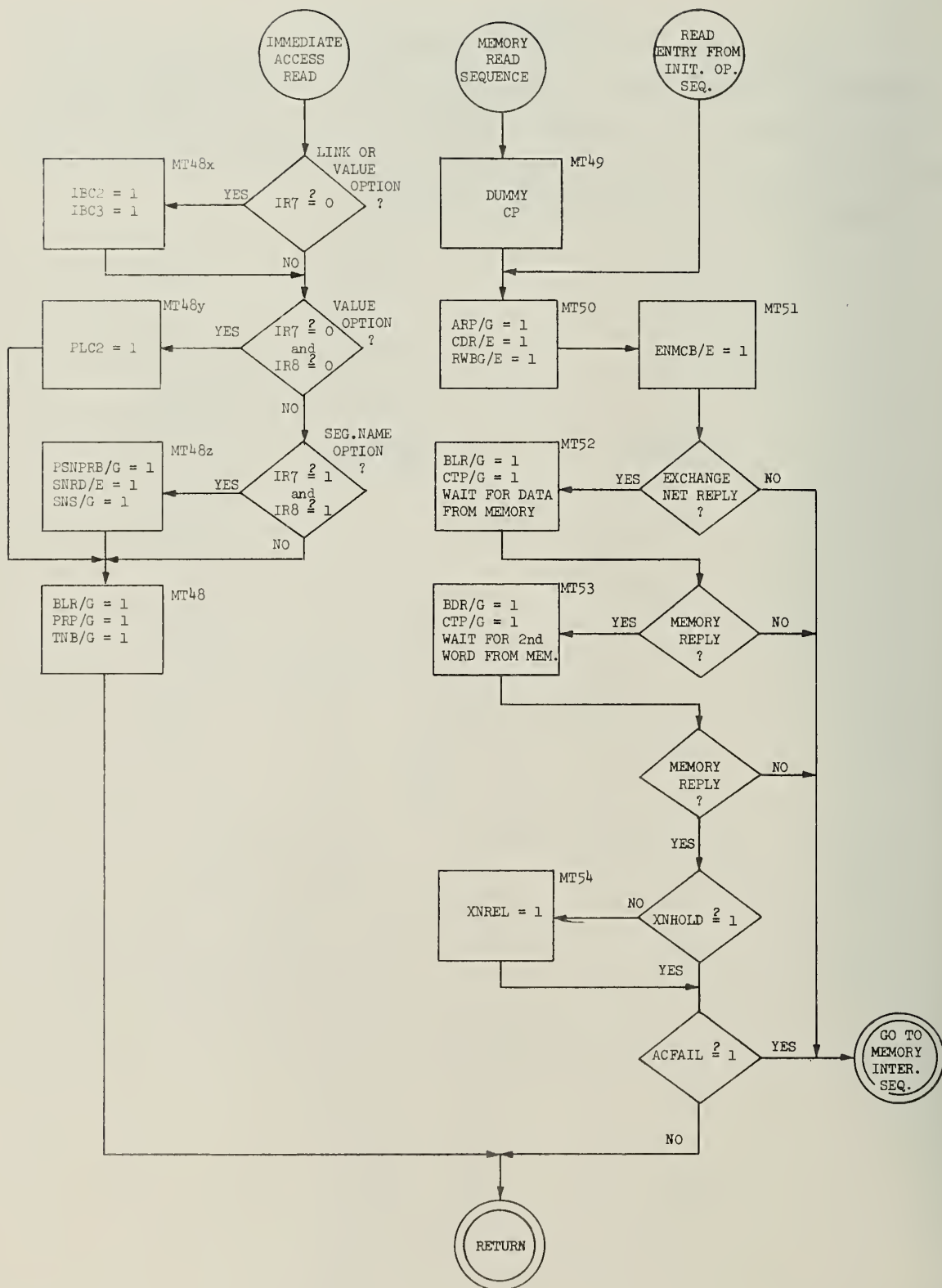


Figure 4.2.3.5/3 TP-Exchange Net Interface Logic for Generation of MDAT1, MDAT2, and MDATRT Signals

After the second data return the Exchange Net will be released provided that XNHOLD does not equal '1'. Then the ACFAIL line must be checked to make sure that the access did not have a memory malfunction or parity error. If it did not the control sequence returns but if it did the memory interrupt sequence must be started.



Memory Sequence - Memory READ  
Control Point Flow Chart

#### 4.2.3.6 Memory WRITE Sequence Control Logic

As in the case of the READ sequence, the WRITE sequence logic is somewhat unconventional. It is controlled by one control signal, XNHOLD and one control flip-flop, IMF. If XNHOLD is activated the sequence will not release the Exchange Net when it is finished. If IMF is set to 1, the WRITE sequence will perform an immediate access write.

The purpose of an immediate access write is to store data in the pointer register fields of the PR indicated by the tag register instead of writing into core at the location specified by that PR. In this case the WRITE sequence will begin at M55 if the register, value, or link options have been specified or at M56 if the segment name option has been specified. In either case the data which is stored in the LR will be gated into the proper PR field(s).

The actual gating in an immediate write depends on the option. If the register option is selected, both the link and value fields are gated. If the link or value options are used, there will only be a half-word of data left justified in the LR. In this case the respective field is gated but the flags are inhibited. In the value option the data must be permuted 2 bytes to the left. In all the link, value and register cases, the data is gated from the LR to the PR selected by the tag register. In the segment name option the halfword left justified in the LR is gated to the segment name register selected by the contents of the tag register.

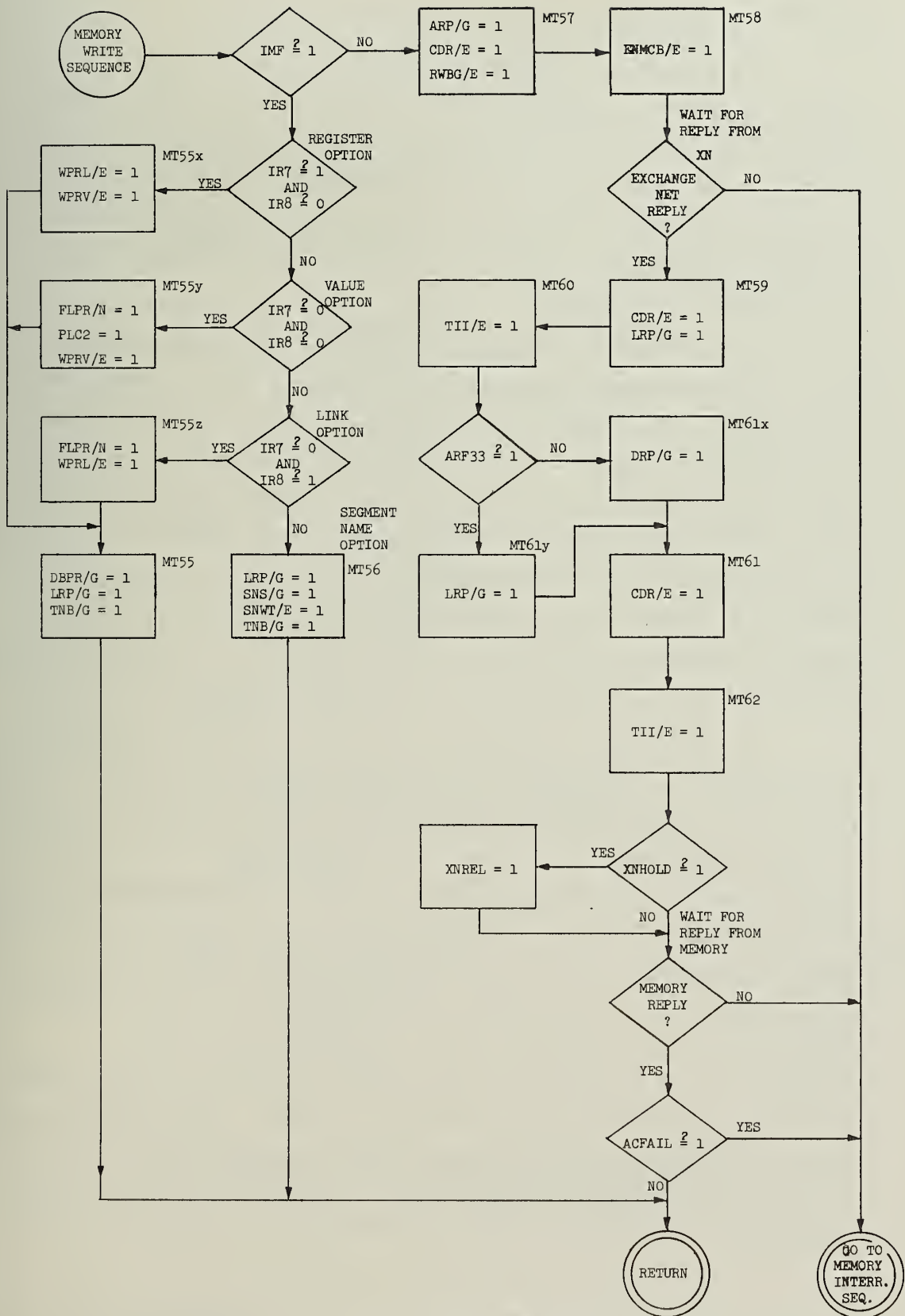
If the memory write access is not an immediate access the sequence will begin with complex control point M57-M58, which requests a path to the memory units from the Exchange Net. Once an open path has been obtained, the WRITE sequence must immediately begin sending the data to be written. There are two time slots during which data is sent to the memory unit. If the address of the cell being written into begins in the second half of the double word, the data must be sent during the second time slot. Otherwise it must be sent during the first. Thus, in order to save logic, on any cell beginning in the second half of the double word, the contents of the LR are sent during both time slots. In all other cases, the LR is sent on the first time slice and the DR on the second time slice. This method works, mainly because the memory unit will ignore all unnecessary data information (based on the status of the Read/Write byte which was sent

when the Exchange Net was first accessed).

The data is actually sent by the M59-M60 and M61-M62 complex control points. These complex control points are made up of standard control points plus extra delay logic. The delay elements within M59 and M61 are set to allow time for the data to be gated from the selected register to the cable drivers. The advance out signals are then used as MT60 and MT62 and are made to turn on the TII signal. The delay elements following these signals are set to keep the TII signal on long enough to satisfy the requirements of the memory unit.

After both words of data have been sent, the WRITE sequence will release the Exchange Net provided that XNHOLD is not on. The sequence must wait for a return from the memory unit indicating that the data has been received. The Exchange Net-TP Interface logic generates the MURT signal for this purpose. The waiting is performed by a calling control point which uses MURT as the normal return and MUNRP, the memory unit no reply signal, as the interrupt signal. As in the Read Access logic the interrupt out signal is not connected since the Exchange Net-TP Interface Logic will generate the interrupt if the memory unit does not respond within the specified time.

After the memory unit has replied the ACFAIL line is checked to see if there was a parity error in transmission or a memory unit malfunction. If so, control is transferred to the Memory Interrupt Sequence. Otherwise, the sequence returns.



Memory Sequence - Memory WRITE  
Control Point Flow Chart



#### 4.2.3.7 Memory Access Interrupt Sequence Description

The purpose of the Memory Access Interrupt Logic is to prepare the TP for a memory access interrupt return if any interrupt conditions are detected during the execution of the memory access sequence. The interrupt operations can be divided into two stages: setting the interrupt indicator logic, and saving the necessary interrupt information so that the interrupt processor will be able to determine what happened.

The interrupt indicator is set at the time the interrupt is first detected by activating one of the special control lines used for this purpose. The table shown in Figure 4.2.3.7/1 lists the various interrupts which might occur and groups them into sets of temporally exclusive interrupts, i.e. interrupts which cannot be on at the same time. This allows them to be coded using a fewer number of bits than if each condition had its own indicator. The logic in Figure 4.2.3.7/2 depicts the design of the relevant part of the interrupt indicator logic and shows how the indicator setting signals in the memory sequence are used to set it properly, according to the codes given in Figure 4.2.3.7/3.

The process of saving the necessary interrupt information is somewhat different in the memory sequence than it is in the other TP sequences. This is because in the case of the memory sequence, unlike the other TP sequences, the important interrupt information may be lost once the sequence interrupt return is activated. Thus it becomes necessary for the memory sequence itself to save the interrupt information instead of waiting for the main interrupt sequence to do so. In particular, it must store the segment name and virtual address of the particular access which caused the interrupt. As a convenience to the calling sequence, it also leaves the virtual address in the DR before it returns. This, in many cases, allows the calling sequence to undo what it had previously done before the interrupt occurred, and thus "buck" the interrupt still further "upstairs".



1	2	3	4	5
---	---	---	---	---

# 1) Hardware/Software Error

## Hardware Errors:

- 2-3) 00 - Memory Unit
  - 01 - Arithmetic Unit
  - 10 - Pattern Articulation Unit
  - 11 - Interrupt Unit

- 4-5) 00 - No Interrupt
  - 01 - Parity Error
  - 10 - Unit Malfunction
  - 11 - No Reply

## Software Errors:

- 2) Segment/Segment Name Table Error
- 3) Not There Interrupt/Other Interrupt

### Not There:

### Other:

- 4-5) 00 - no interrupt
  - 01 - Page Map
  - 10 - Data
  - 11 - Page

- 4-5) 00 - No Interrupt
  - 01 - Bounds Overflow
  - 10 - Trap
  - 11 - Illegal Access

Figure 4.2.3.7/1 Arrangement of Interrupt  
Indicator for Memory Interrupts

<u>IN1</u>	<u>IN2</u>	<u>IN3</u>	<u>IN4</u>	
0	0	0	0	No Indicator
0	0	0	1	STBOV
0	0	1	0	STTRP
0	0	1	1	SILAC
0	1	0	0	No Indicator
0	1	0	1	STMNT
0	1	1	0	STDNT
0	1	1	1	STPNT
1	0	0	0	No Indicator
1	0	0	1	SGBOV
1	0	1	0	SGTRP
1	0	1	1	SILAC
1	1	0	0	No Indicator
1	1	0	1	SGMNT
1	1	1	0	SGDNT
1	1	1	1	SGPNT

Figure 4.2.3. /3 Indicator Codes for Memory Interrupts

There are actually three possible situations which may occur when an interrupt situation is detected in the memory sequence.

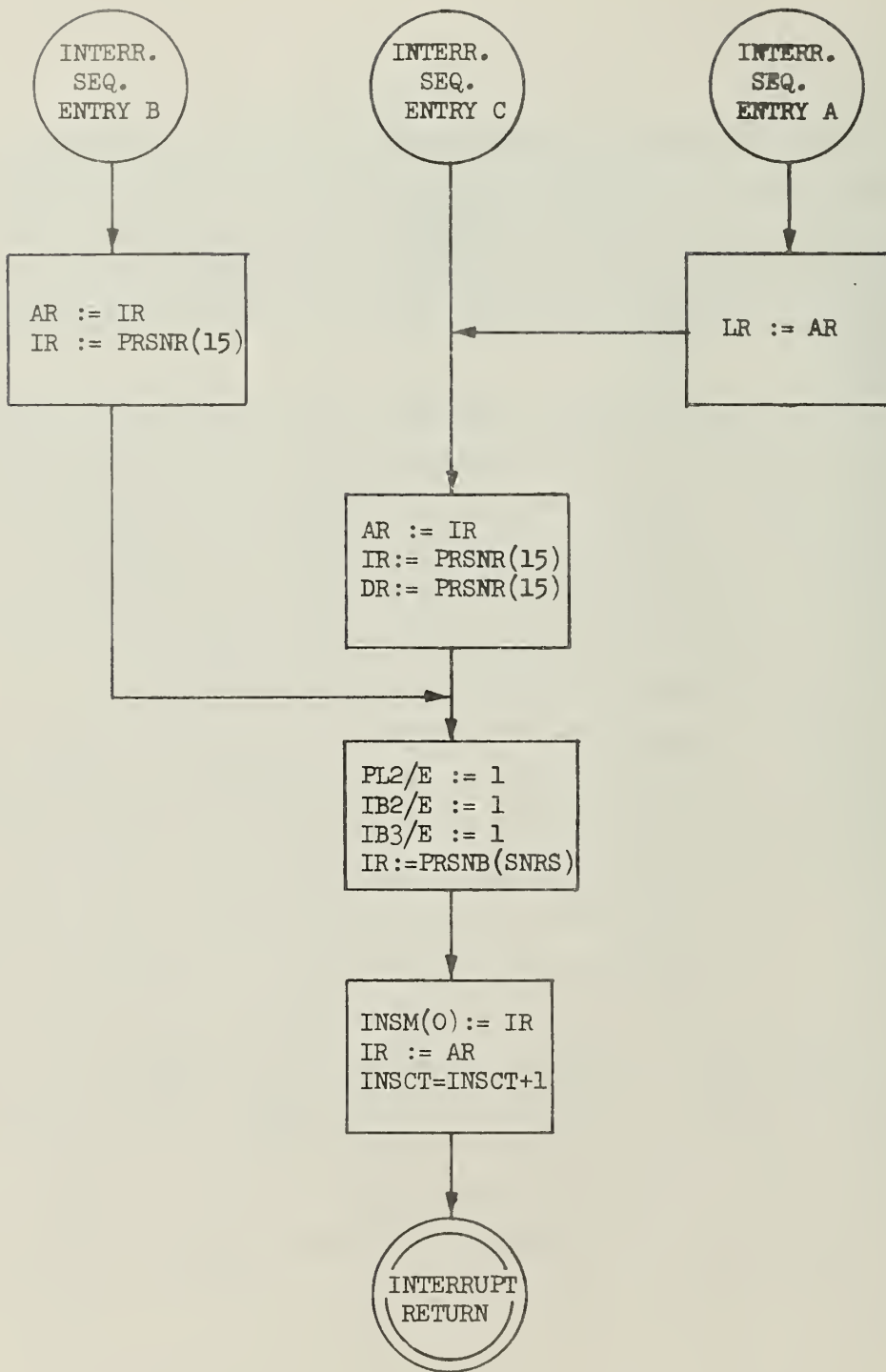
(a) If an interrupt is detected in the Partitioned Mode sequence, the AR will contain data which must eventually be stored in the LR before making an interrupt return, and the DR and LR will contain junk.

(b) If an interrupt is detected in the Memory Write sequence, both the LR and DR will contain data which must be saved while the AR will contain junk. In this case, the DR is not to be loaded with the virtual address of the access (which would still be in PRSNR(15)) since the DR may contain data which was to have been written into core.

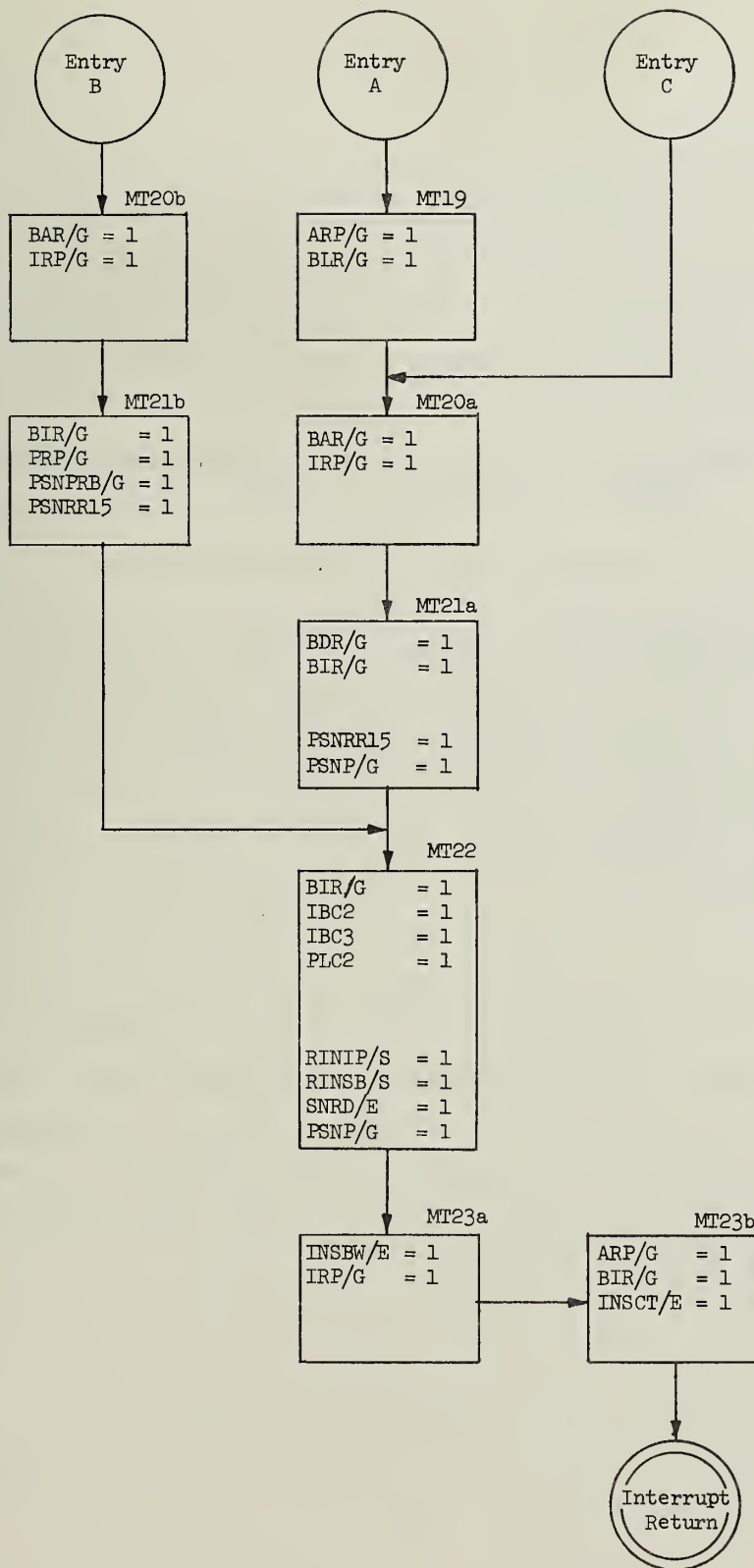
(c) In all other memory interrupt situations the AR and DR will contain unneeded information which must be saved.

Thus the memory interrupt sequence is set up as follows: its main task is to construct a one word datum containing the segment name and the virtual address and store it in the interrupt storage memory. This can be most easily done using the merging options in the IR. Before this can be done, however, the current contents of the IR must be preserved. The natural place to do this is in the AR since, except for the Partitioned Mode case, it always contains junk and even in this one exception its contents are supposed to be transferred to the LR anyway, after which its data is no longer needed. In those cases where the DR must be loaded with the virtual address, this operation is performed before the merging takes place.

A flowchart for the Memory Interrupt Sequence is given at the end of this section.



Memory Sequence - Memory Interrupt



Memory Sequence - Memory Interrupt Control Step Flow Chart

### 4.3 Pointer Stack Operations

The Pointer Stack operations, as stipulated by the slashing conventions (see Section 4.4), are used during the processing of the various instructions. The format for the Pointer Stacks is shown in Figure 4.3. Each cell consists of a double word which is subdivided

LINK	VALUE	NAME	not used
------	-------	------	-------------

Figure 4.3 . Pointer Stack Format

into 4 halfword fields. The first halfword is used for the link address to the next lowest entry in the stack. The second and third halfwords contain the Pointer Register value and segment name respectively, preserving the contents of the Pointer Register when this particular cell was pushed into the stack. These three halfwords are in the normal format as used for a Pointer Register. The last halfword of the stack cell is not used.

The main Pointer Stack operations are STACK and UNSTACK. These operations are used respectively to "push" the current contents of a PR into its stack and to "pop" the top of a stack in core into its PR. To perform these operations it is also necessary to call available space sequences which get cells from available space as they are needed or return them when they are no longer needed. These sequences are called AS GET and AS RESTORE respectively.

#### 4.3.1 Available Space Sequences

##### 4.3.1.1 Available Space Sequence Descriptions

The available space sequences control the use of the available space both in Pointer Stack operations (where PR#14 is used) and in GET and PUT instructions where the program has declared one or more PR's to be in the available space format. The available space format is explained in Section 2.4.1.3. There are two available space sequences: AS GET and AS RESTORE.

These sequences are called most frequently by the TP to manipulate pointer stacks. In these instances PR#14 serves as the source of the LINK and COUNT quantities. For GET and PUT instructions, however, the TGR identifies the available space pointer. Thus there are two possible sources for the PR name: the TGR or the NPR14/S signal which forces the selection of PR#14 by the name bus. In the Available Space control logic this selection is made by a special flip-flop, ASTRS (Available Space Tag Register Select), which must be set (1 for tag register, 0 for PR#14) before entry into the sequence. This flip-flop then controls the source for the name which appears on the name bus when an available space pointer register must be referenced. In the flowcharts the controlling pointer register is indicated by PR(NB) where it is understood that ASTRS will control what is on the name bus.

The AS GET sequence is used to obtain a cell from the "free list" in the available space file. If the "free list" is empty the sequence will try to get the cell from the consecutive storage area. If this in turn is empty, an Available Space Empty interrupt is generated.

If the AS GET sequence takes its cell from the "free list", it must use the link in the available space pointer register to access the



top free list cell in order to find out what that cell is pointing to. This new link is then placed in the left half of the available space pointer register and the address of the previous top cell of the free list is left in the DR (right justified) for use by the "calling sequence."

If the free list is empty the cell will be taken from consecutive storage in one of two possible ways. If the cell is to be taken from PR#14 available space then the cell size is assumed to be a double word. In this case 8 is automatically added to the count. Before this is done, however, the old count is gated to the LR to be saved. If the available space PR is determined by the Tag Register (i.e. if ASTRS = 1), then the cell size may be any number of even bytes up to 256. The exact number is contained in the first halfword of the segment. Thus in this case the DR is set to 0 and then used to access this first halfword. On return from the memory sequence, the LR will contain this halfword integer, left-justified. After gating it into the DR, right-justified, the DB can be loaded with the available space pointer register count and the cell size can be added to it. At the same time that the addition is being performed the LR can be loaded from the DB with the old count.

Regardless of how the new count is obtained, it is stored in the DR. The next step is to make sure that the new beginning of the consecutive storage is still within the part of core allowed for the available space file. If it is not, then the cell we just obtained (its address is the old count) may not be entirely within bounds and an Available Space Empty interrupt is made.

The check for Available Space Empty is made by performing a bounds check on the new beginning address of the consecutive storage. If the check finds that the obtained cell is within bounds, the new count, which was stored in the DR, is loaded into the link and count fields of the Available Space PR. Then the LR, which has previously been used to store the old Available Space count field, is gated to the DR and the sequence returns.

To determine if the free list is empty, the link and count fields of the available space pointer register are compared to see if they are equal. If they are, the free list is empty. The logic used to make the comparison is shown in Figure 4.3.1.1.

The flip-flop CONS is set to one whenever a cell is obtained from consecutive storage instead of the free list. This flip-flop may be used by sequences which call AS GET if it ever becomes necessary, due to an interrupt, to undo the effects of the AS GET sequence at some later point in the sequence which called it. Since any interrupt which originates within AS GET will occur before any permanent changes in the registers are made, no special operations are necessary to "undo" anything.

The AS RESTORE sequence is used to return a cell to the available space free list. The 16-bit address of the cell to be returned is placed right-justified in the DR by the "calling sequence". The AS RESTORE sequence then accesses this cell and stores in it the link currently residing in the available space pointer register. It then takes the address of the cell being returned and stores it in the link of the available space PR. In summary the available space pointer register now points to the cell which was just added and that cell, in turn, points to the cell which was previously the top cell of the free list.

Note that if an interrupt occurs in the AS RESTORE sequence during the address construction phase of the memory sequence, the DR is restored to its original value by the memory interrupt sequence before the interrupt return is performed. This enables whatever sequence called AS RESTORE to get rid of the cell ultimately.

The detailed flowcharts for both of these sequences are at the end of this section. Figures 4.3.2.1/1 through /3 of Section 4.3.2.1 gives pictorial examples of Pointer Stack Operations which should help explain the operation of AS GET and AS RESTORE.

LKCTC/E

DB35  
 $\overline{\text{DB17}}$   
 $\overline{\text{DB35}}$   
 $\overline{\text{DB17}}$   
 $\overline{\text{DB34}}$   
 $\overline{\text{DB16}}$   
 $\overline{\text{DB34}}$   
 $\overline{\text{DB16}}$

DB20  
 $\overline{\text{DB2}}$   
 $\overline{\text{DB20}}$   
 $\overline{\text{DB2}}$   
 $\overline{\text{DB19}}$   
 $\overline{\text{DB1}}$   
 $\overline{\text{DB19}}$   
 $\overline{\text{DB1}}$

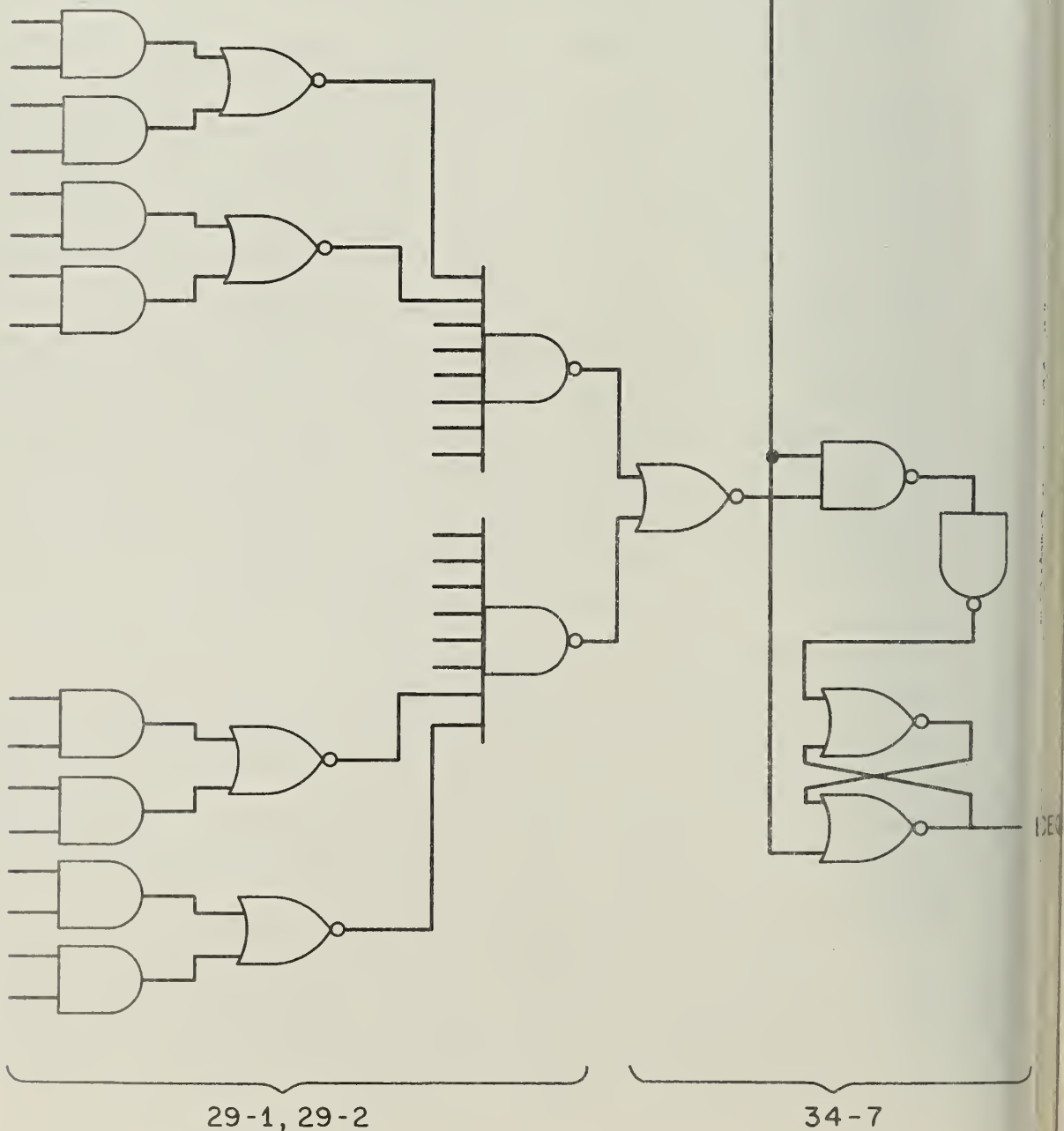
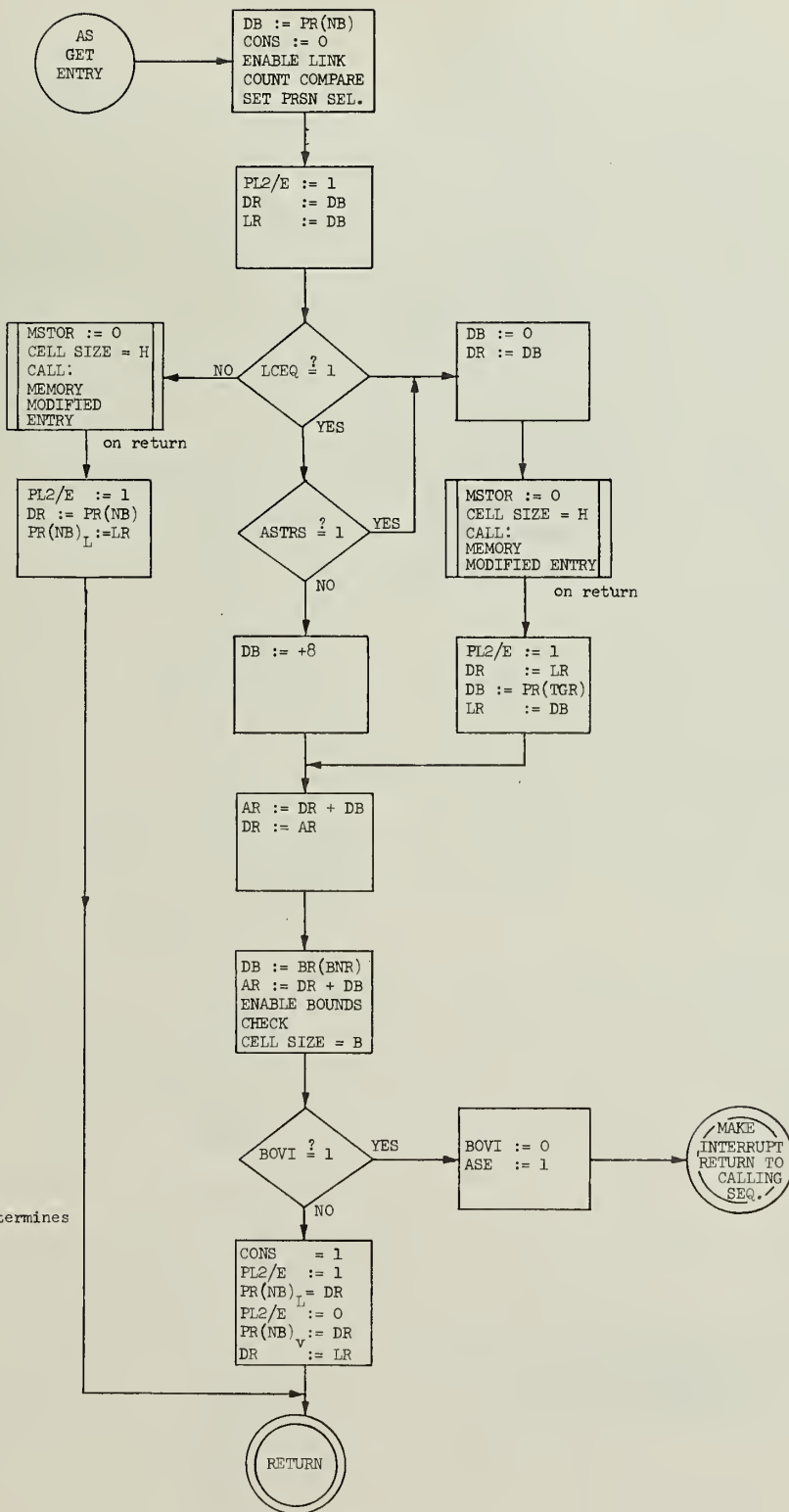
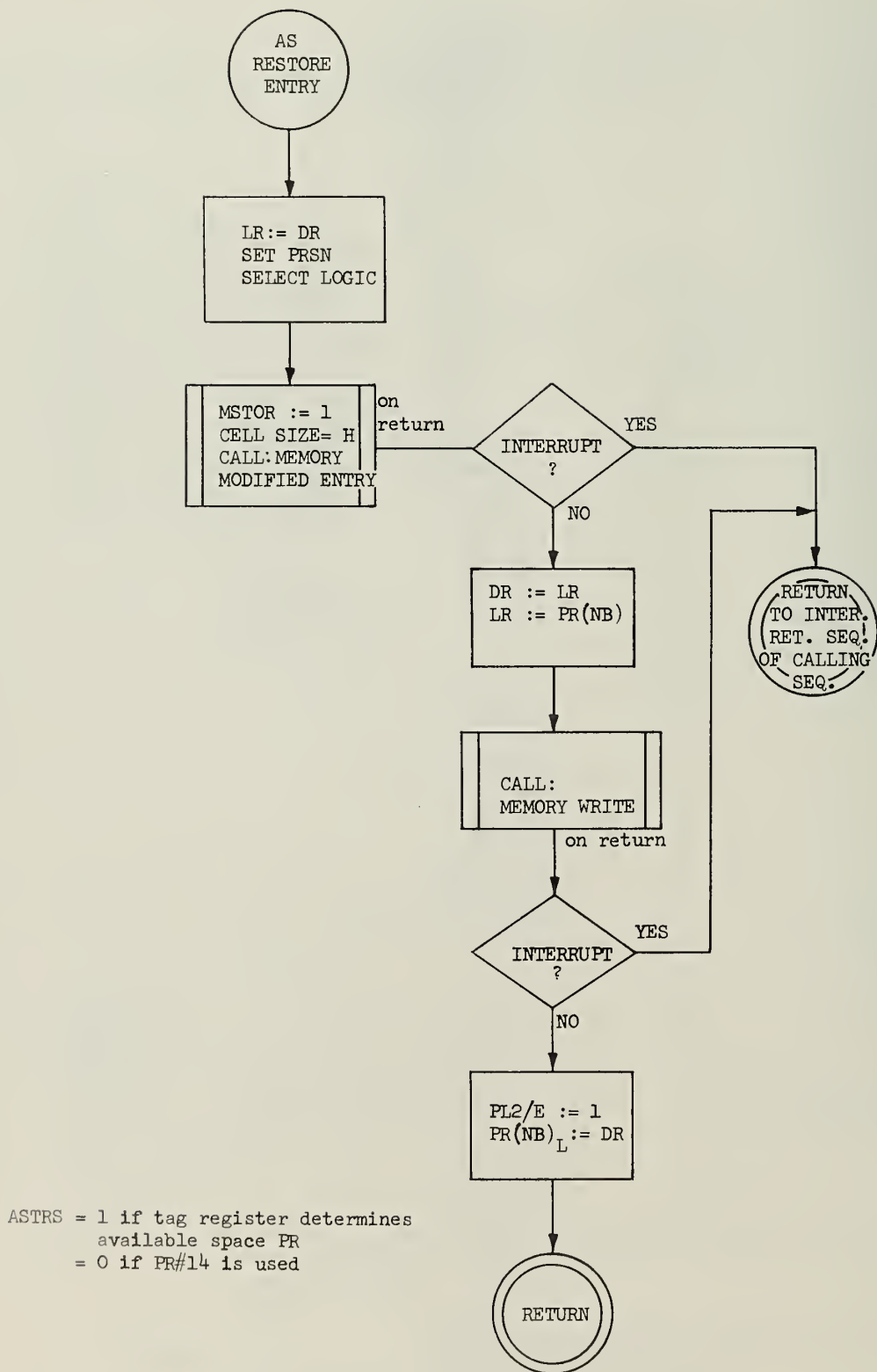


FIGURE 4.3.1.1 IC IMPLEMENTATION OF LINK-COUNT COMPARE

ASTRS = 1 if tag register determines  
available space PR  
= 0 if PR/14 is used



Pointer Stack Available Space Sequence Flow Charts  
AS GET Sequence



Pointer Stack Available Space Sequence Flow Charts  
AS RESTORE Sequence

#### 4.3.1.2 AS GET Control Logic

The AS GET sequence obtains a cell from the available space list and returns the address of this cell in the DR right justified. The only control flip-flop is ASTRS, the Available Space Tag Register Select flip-flop, which is shared with the AS RESTORE control logic. It must be set to "0" if PR#14 is to be used to access the available space file and set to "1" if the file will be indicated by the PR whose name is in the tag register. The flip-flop controls which input is selected for the NAME BUS: TNB/G if ASTRS = 1 and NPR14/S if ASTRS = 0.

Most of the logic is fairly straightforward. AST2 is shared between two control steps in the AS GET sequence.

Note that the cell size is controlled by using the direct control lines. The desired line is held on during each memory access and during the addition. Thus no change is made in the status of the cell size flip-flops during the sequence. However, the Cell Size Selector is changed and must be changed back by the calling sequence if it is necessary to use some other selection. This same technique is used in the AS RESTORE sequence.

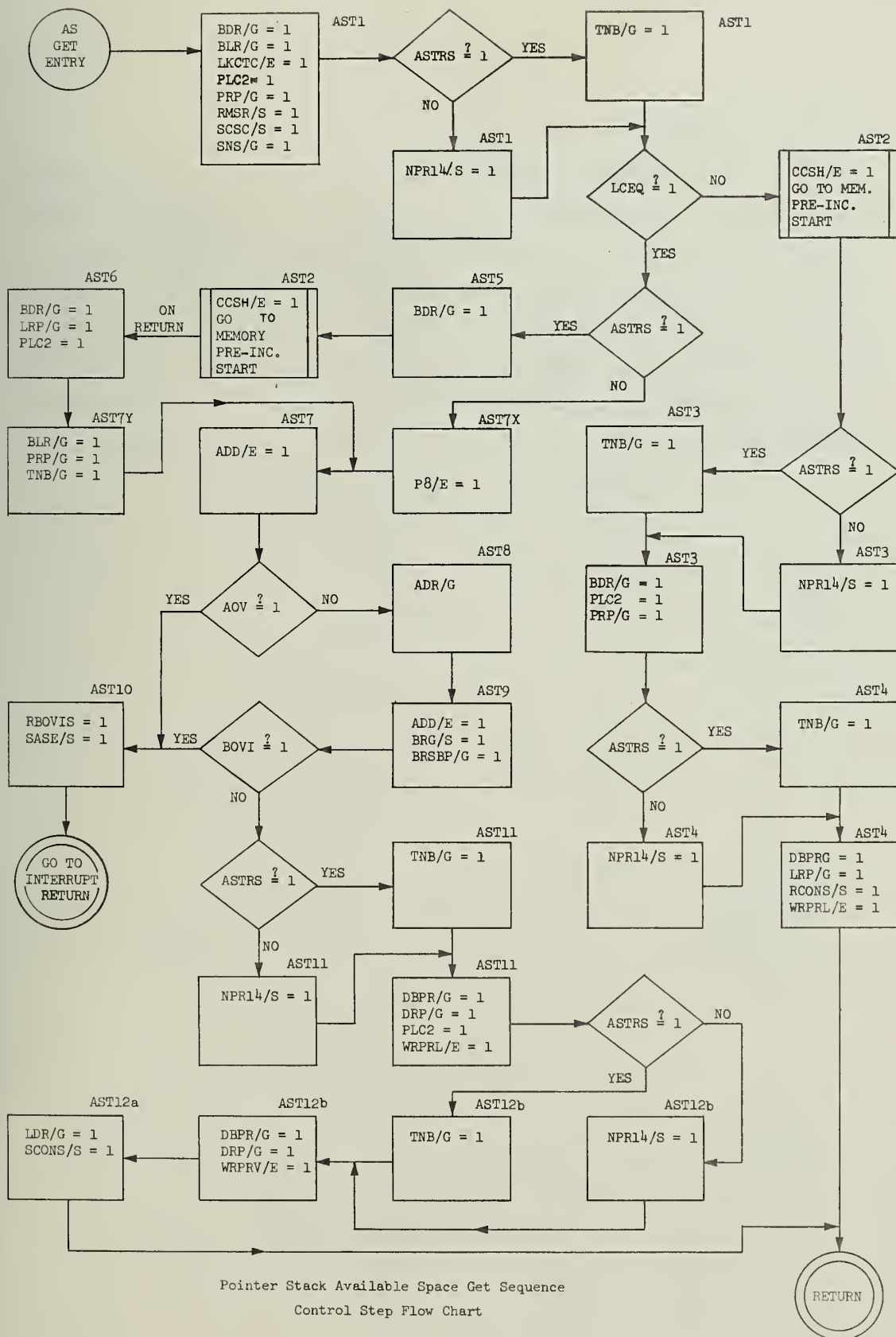
When a new cell must be obtained from contiguous storage in the AS GET sequence, the 32-bit Adder is used to increment the count. Note that the inhibits are not turned on for the leftmost two bytes in the result. There is no real need to do this since when the new count is gated back to the available space PR, these two bytes will not be used.

If this two byte addition produces an overflow, the resulting address will "wrap around" to the low end of the segment. This can cause problems since if the bounds overflow test were made on this new address, no violation would be detected. Thus after the addition, if an adder overflow is detected, an Available Space empty interrupt return is executed.



A minor problem can arise in using the bounds check on the new count to determine if there is space remaining in the Available Space segment: if the last cell ends exactly on the upper boundary, the bounds check of the new count will indicate an overflow even though there is, in fact, one cell left. This occurs because the bounds check logic checks the validity of the one byte cell beginning at the new address. In order to fix this it would only be necessary to subtract 1 from the count before checking its bounds. However, it was felt that this solution was more trouble than the fact that one cell might be wasted, so it was not used.

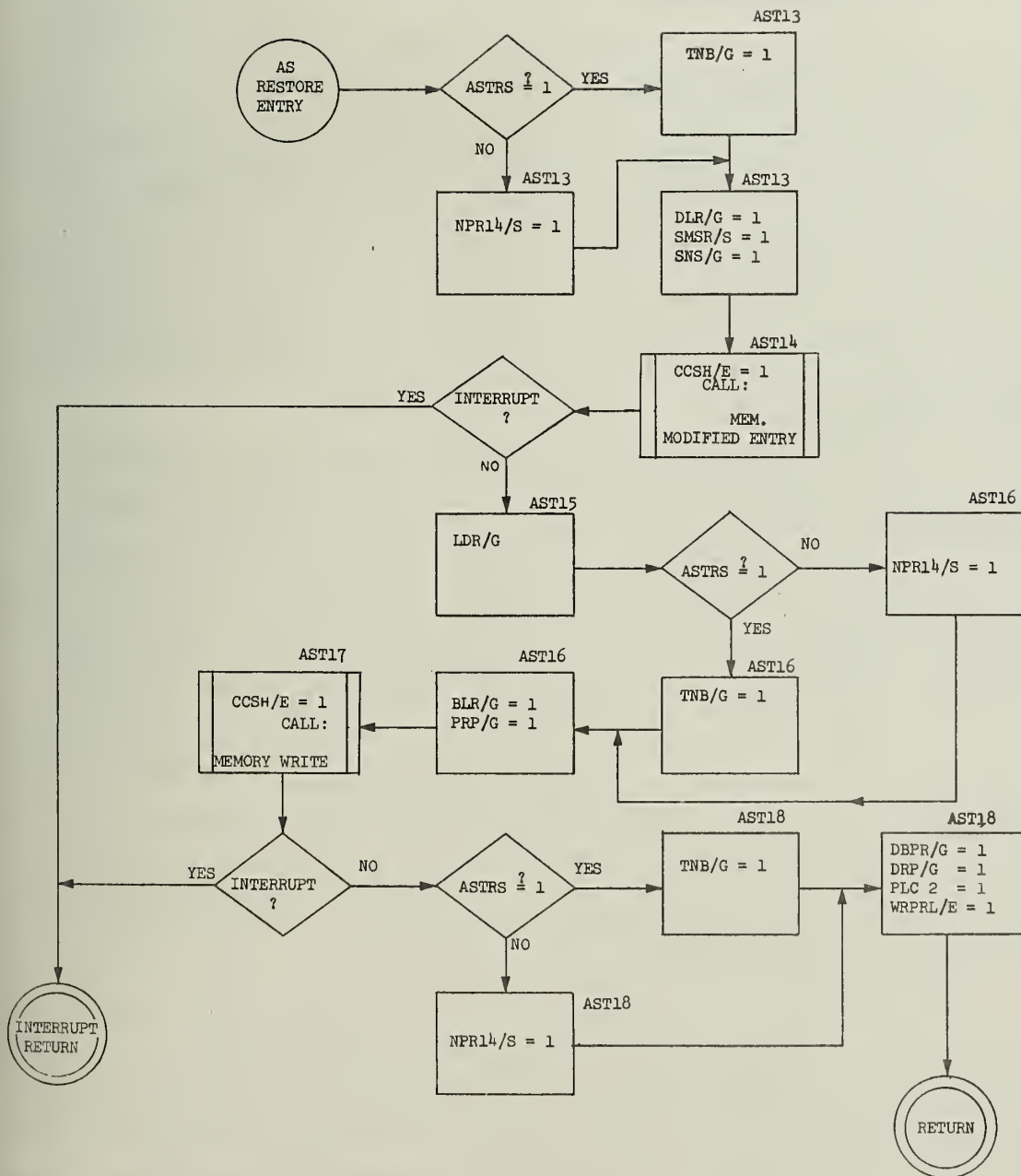




#### 4.3.1.3 AS RESTORE Control Logic

The AS RESTORE Control Sequence is used to return a cell, whose address is contained right-justified in the DR, to the free list of a given available space pointer register. It utilizes one control flip-flop, ASTRS, the Available Space Tag Register Select flip-flop, which is also used by the AS GET sequence. This flip-flop, if on, indicates that the TGR is to be used as the source for the name of the available space pointer register. If ASTRS = 0, PR#14, the available space pointer register for the PR stacks, is used.

The sequence itself is quite straightforward.



Pointer Stack Available Space Sequence  
AS RESTORE Control Step Flow Chart

### 4.3.2 Pointer Stack Sequences

#### 4.3.2.1 Pointer Stack Sequence Descriptions

As described in Section 4.3.1, these sequences are used for stacking and unstacking pointer stacks.

The STACK sequence, as shown in the flowchart at the end of this section, consists of first using AS GET to obtain an empty cell from available space. The 16-bit address of this cell is stored by AS GET right-justified in the DR. It should be noted that in this case, as in every case of an address (or link) being used from the pointer register available space file, the base register named by PR#14 is used in addressing memory.

Next the cell address in the DR is copied into the LR for storage while a memory write is initiated. If an interrupt occurs during the address construction phase of the memory sequence then the cell obtained by the AS GET sequence must be returned. This can be done without accessing core since that cell still contains the links which connect it with either the free list or consecutive storage. Thus if CONS = 0 the cell was obtained from the free list and only the available space PR link field need be loaded with the address of the obtained cell in order to get it back on the free list. If CONS = 1 the cell was obtained from consecutive storage and both fields of the available space PR will have to be loaded with the address of the obtained cell in order to put it back in consecutive storage.

If there was no interrupt during the address construction phase of the memory sequence, then the STACK PR sequence continues by loading the DR (left-justified) with the segment name of the PR which is being stacked. Then the name is merged into the LR so that the segment name is in the left halfword of the LR and the address of the new cell being accessed is in the right halfword. Finally the LR is gated to the DR and then reloaded with the contents of the PR to be stacked. Control then

returns to the memory sequence which writes the data into the cell obtained by AS GET.

It should be noted that a double word access is made when the PR information is being stacked. However, the three PR fields (link, value and segment name) take up only six bytes. This means that the last 2 bytes in the double word cell are filled with whatever happens to be in the right half of the DR. At the present time this is the address of the cell being accessed. This may or may not be of any use. Once the PR has been stacked, however, the data can be overwritten if desired.

When the memory access has been completed, another check for interrupts must be made since there may have been a hardware error in writing the data into core. If there is an interrupt the same Available space pointer restorations as previously described are initiated. If not the PR which was stacked is updated by loading into its link field the address of the cell just stacked. This address is still located in the right half of the DR. After this has been done the STACK PR sequence returns. An example of a stack operation is shown pictorially in Figures 4.3.2.1/1 and 4.3.2.1/2.

The UNSTACK sequence is used to pop off the top cell of a pointer stack. The first step is to load the AR and DR with the pointer link, right-justified. The AR is then checked for zero, since if the link of the PR is zero, there is no more stack and we will not be able to do a pop. If the link is zero, a Stack Empty interrupt will occur.

The next step is to check for a tag of 13. If the tag is 13, the OS must be initialized (if it has not already been so) and the OSC flip-flop must be set. Note that if the OS is cleared, the DR must be reloaded since its contents will have been destroyed during the clearing sequence.

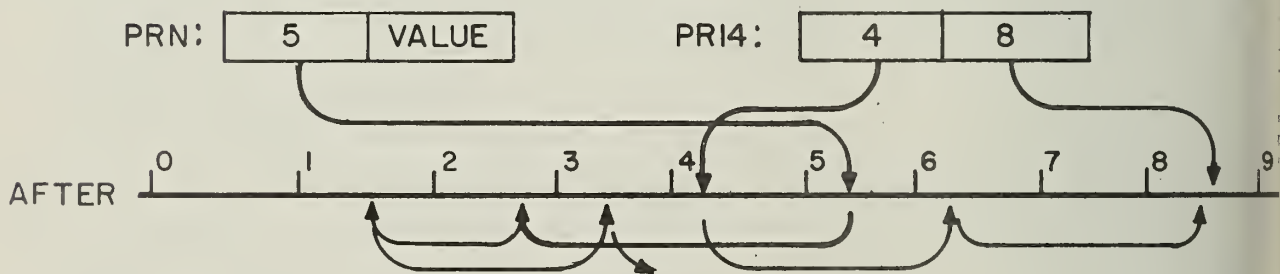
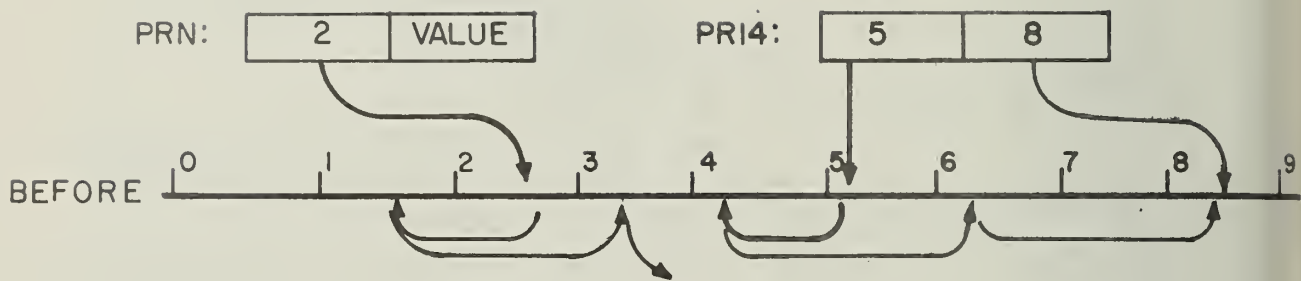


FIGURE 4.3.2.1/1 STACK WITH LINK  $\neq$  COUNT

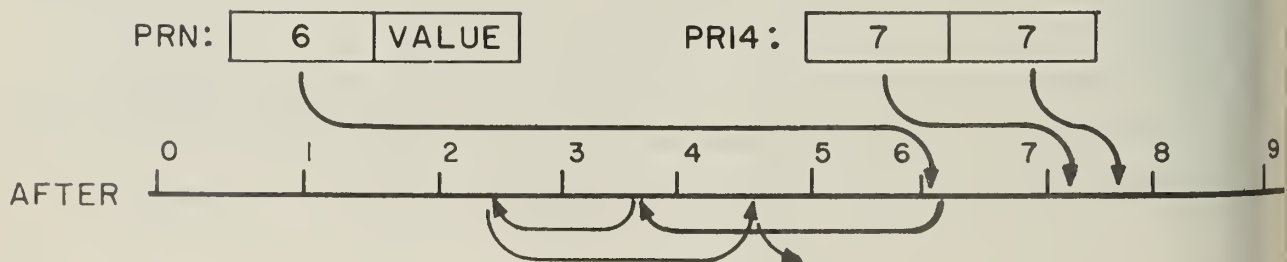
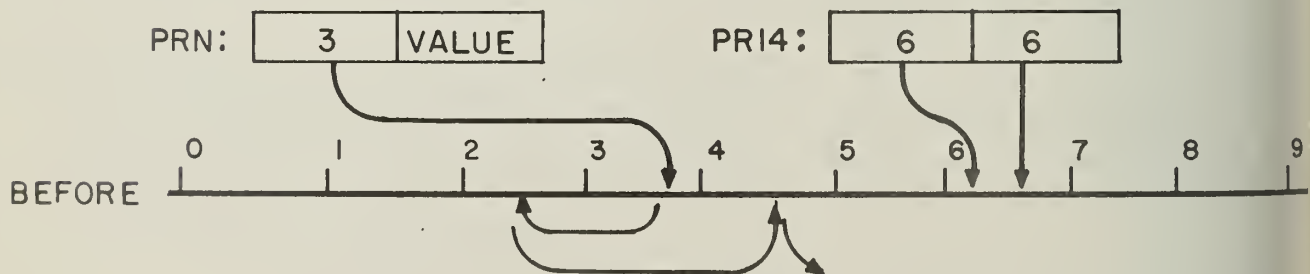


FIGURE 4.3.2.1/2 STACK WITH LINK = COUNT



After the check has been made, the PR Segment Name Register Selector is set to PR#14 and a memory read is performed to obtain the contents of the second PR cell in the stack. This is then loaded into the PR and the cell just accessed is replaced on the free list using the AS RESTORE sequence. Note that after the memory cycle is over, the DR must again be loaded with the address of the cell to be restored. This must obviously be done before the PR is overwritten with the new contents. The DR content is used by the AS RESTORE sequence.

If interrupts occur in either the OS CLEAR or memory sequences, a direct interrupt return can be performed since no changes have been made which would cause the sequence to be non-restartable. However, if an interrupt occurs in AS RESTORE, the contents of the PR will already have been changed to reflect the unstacking and thus this must be undone. The procedure is as follows: load the DR (which during an interrupt of AS RESTORE will contain the address of the cell which was going to be restored to the free list) into the link field of the PR designated by the tag register. This simple action will restore the PR stack to its initial depth before the call of the UNSTACK PR sequence.

It should be noted that the value of the PR will have been changed when the contents of the cell which was going to be restored was loaded into it. There is no way to recover this value. Luckily there is no need to do so. After the interrupt has been processed, if the interrupt is a recoverable one, the TP control logic will continue with the instruction where it left off. This time the cell will definitely be restored to the available space list. Since the top of the PR stack is destined to be thrown away anyway it does not matter that the value field contains garbage.

A pictorial example of UNSTACK is given in Figure 4.3.2.1/3.



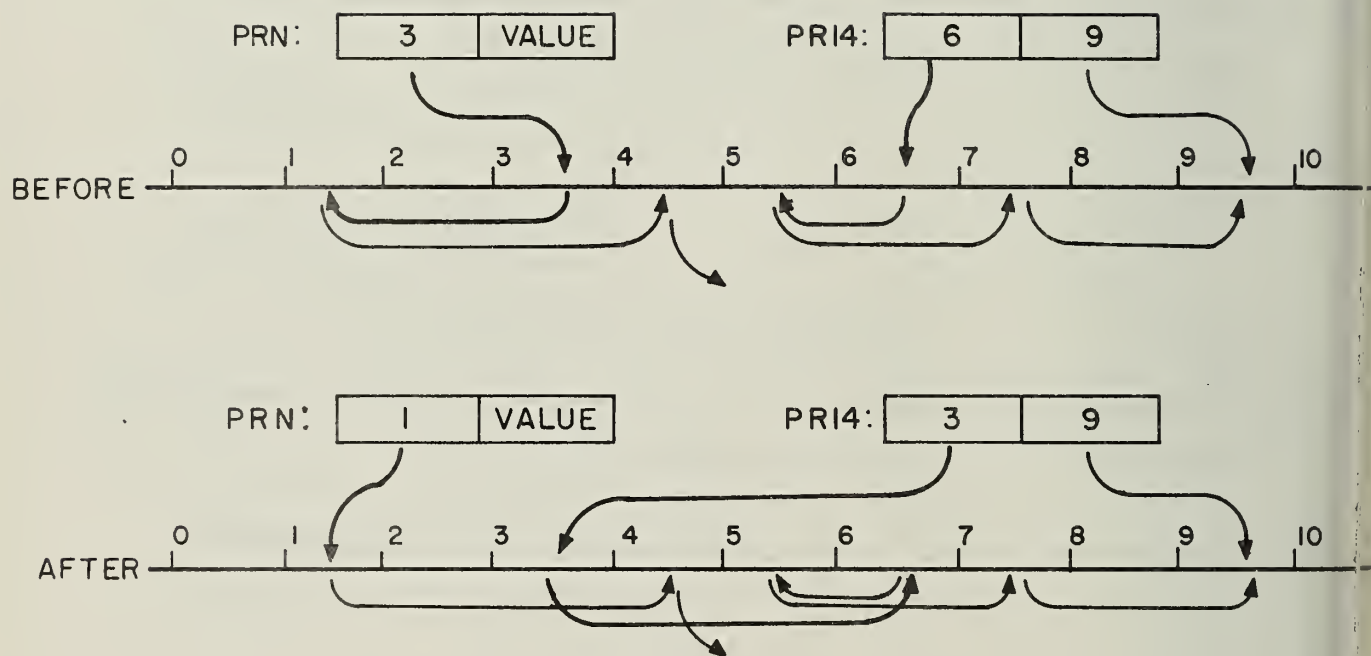
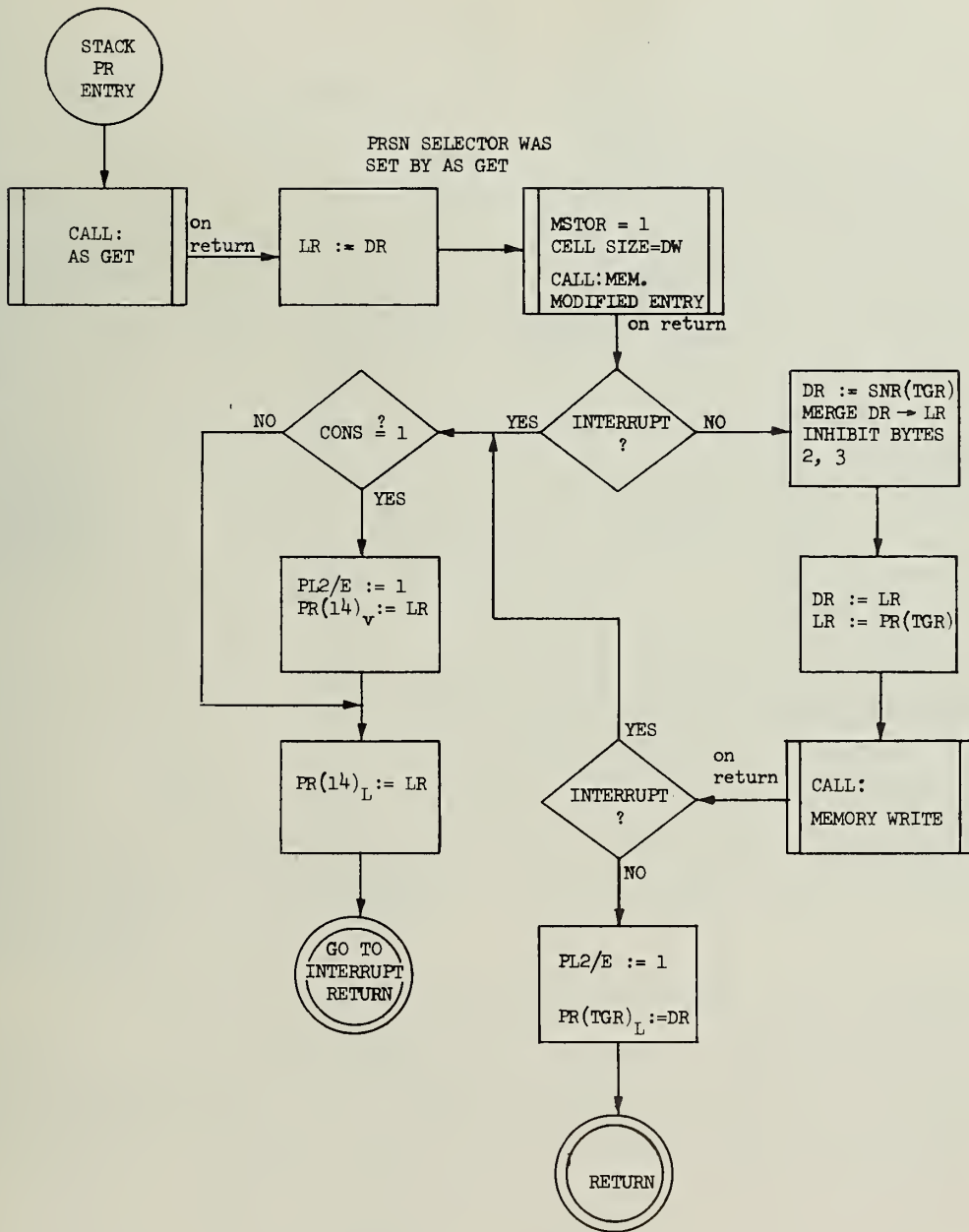
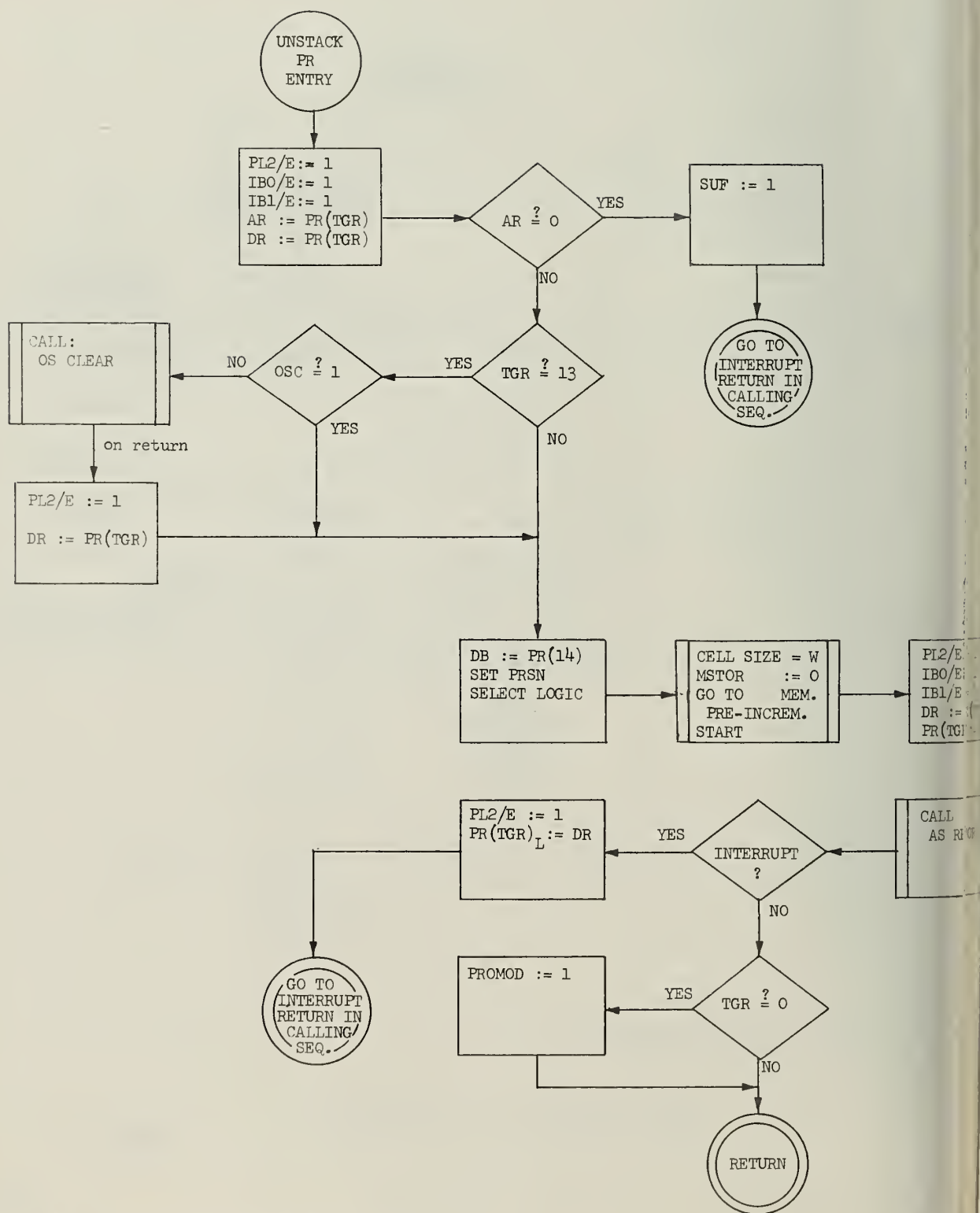


FIGURE 4.3.2.1/3 EFFECTS OF UNSTACK



OSC = 1 if OS has been cleared  
 OSINT = 1 if OS will need to be initialized



OSI = 1 if OS has been cleared  
 OSINT = 1 if OS will need to be initialized

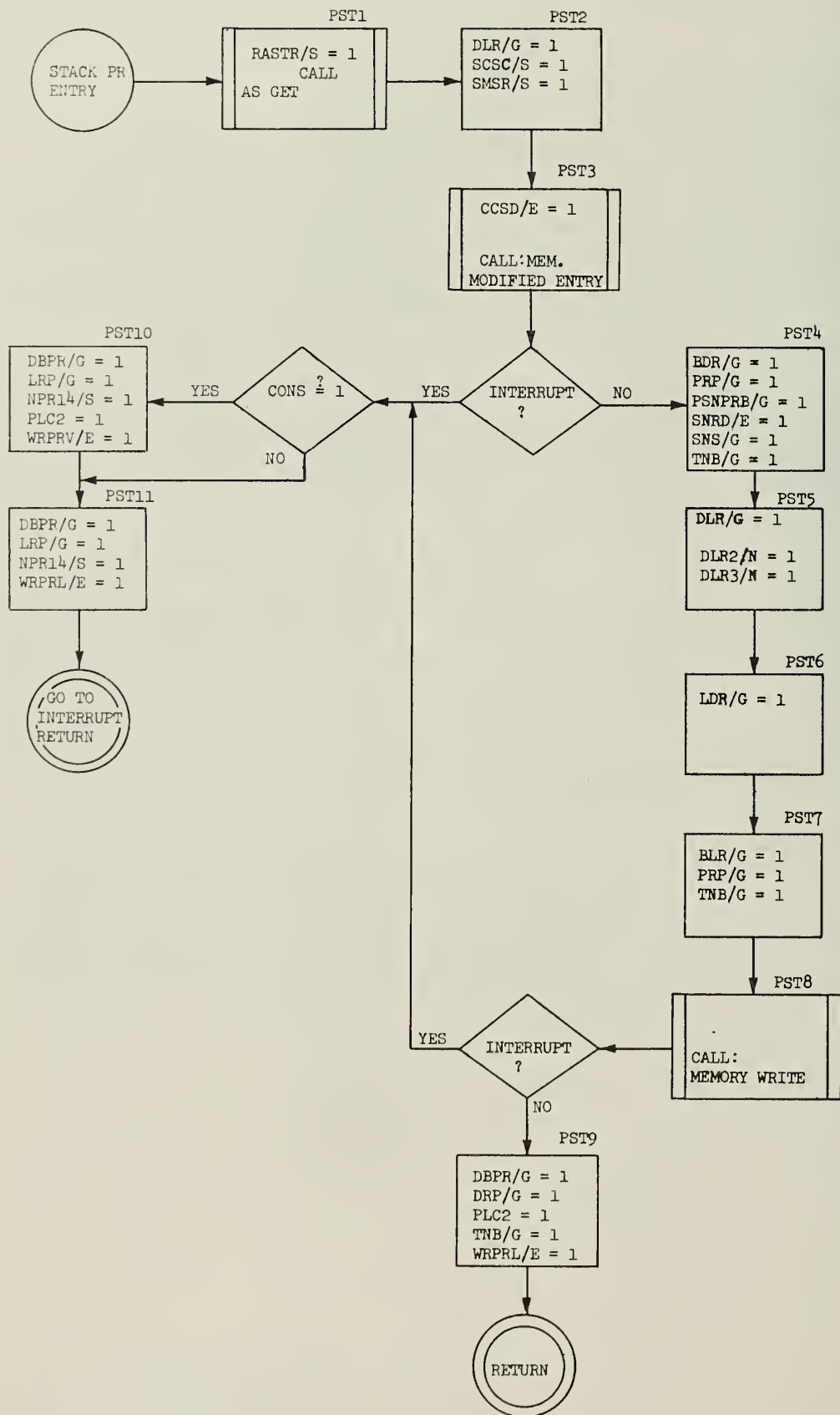
#### 4.3.2.2 STACK PR Control Logic

The STACK PR sequence obtains a cell from available space and then loads it with the current contents of the Pointer Register link, value and segment name fields and places it in the pointer stack immediately "below" the Pointer Register. There are no control signals or flip-flops which must be set before entering this sequence. However, the sequence does make use of the CONS flip-flop, set by the AS GET sequence, if an interrupt occurs while executing the remainder of the sequence.

Note that the STACK PR sequence changes the setting of the cell size selector logic. Thus when using this sequence due care must be taken if the selector was previously set.

Another item to note is the operation of the PRSNR selector. Before the memory access, the selector is set to Segment Name Register #14 by the AS GET sequence. Once the core address has been calculated and the "load registers" signal, LDREG, turned on, this setting is no longer needed. At this point the PRSNR selector is switched to the Segment Name Register corresponding to the PR being stacked, and that segment name can then be stored in the stack in core.

Interrupts during the sequence are handled in one of two ways. If the interrupt occurs during the AS GET sequence, a direct return can be made since the AS GET sequence insures that there are no "loose ends". Otherwise the operations described in Section 4.3.2.1 must be performed before an interrupt return is made.



Pointer Stack Sequences -STACK PR  
Control Step Flow Chart

#### 4.3.2.3 UNSTACK PR Control Logic

The UNSTACK PR sequence is used in adjusting the PR stacks. During its execution the contents of the cell which was originally the second cell in the stack is stored in the PR at the top of the stack. Then this second cell is removed from the stack and returned to available space.

The control logic itself is reasonably straightforward. The Boolean equations for the decision logic after control point PST13 are given in Figure 4.3.2.3.

Note that the direct control lines to the cell size generator are used to specify the cell size for the memory access. This means that the contents of the cell size flip-flops remain unchanged. However, the cell size selector is changed and thus if the calling sequence desires some other selection, it must reset the selector after the UNSTACK PR sequence has returned.

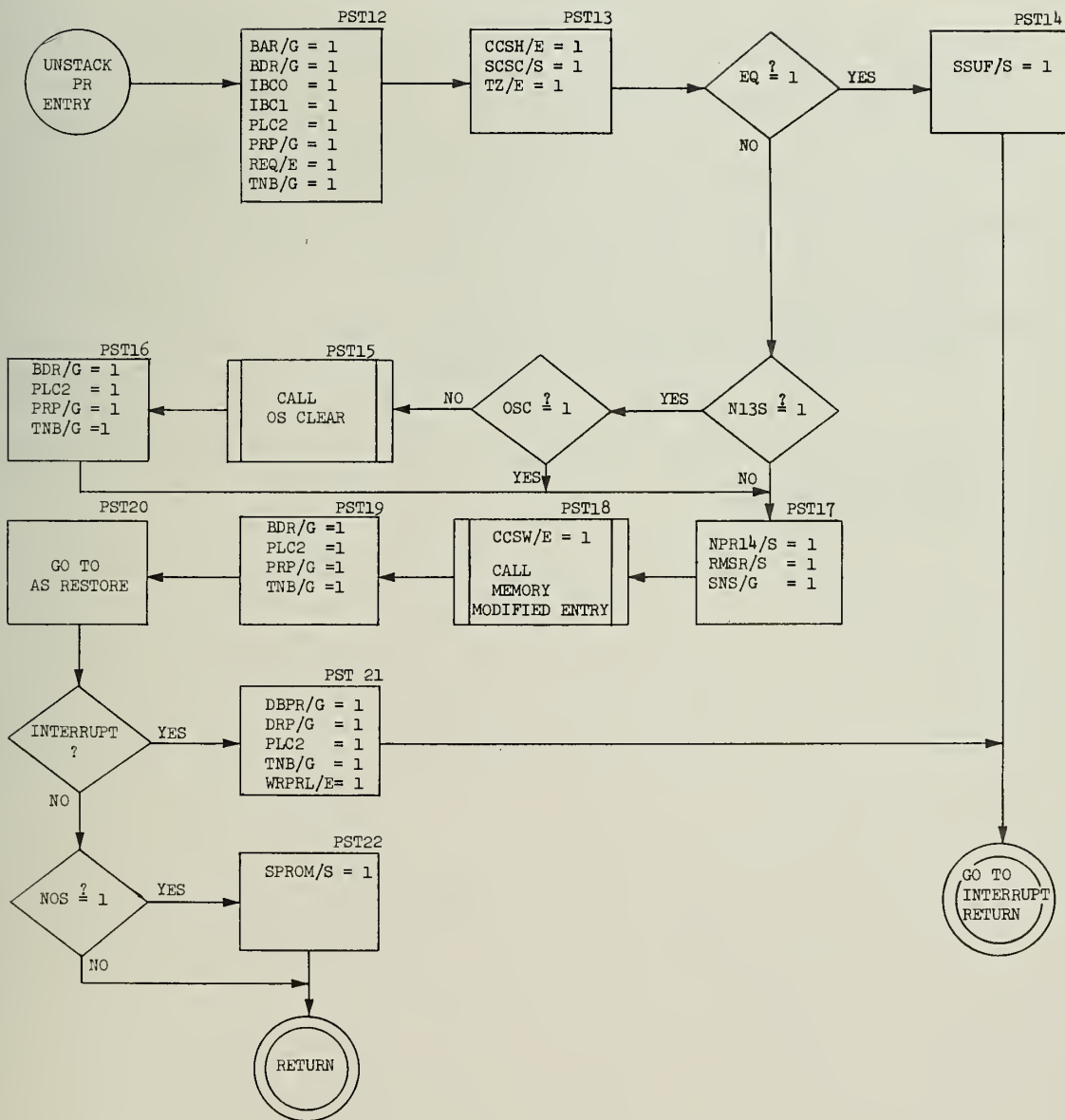
$$PSA\emptyset13 \cdot EQ \quad PST14$$

$$PSA\emptyset13 \cdot \overline{EQ} \cdot N13S \cdot \overline{OSC} \quad PST15$$

$$PSA\emptyset13 \cdot \overline{EQ} \cdot \overline{(N13S \cdot \overline{OSC})} \vee PSA\emptyset16 \quad PST17$$

Figure 4.3.2.3 - Boolean Equations for Decision  
Logic After PST13





Pointer Stack Sequence - UNSTACK PR  
Control Step Flowchart

#### 4.4 Phrase Processing

This section describes the various phrase processing sequences called by the Main Control sequence and its component parts: the Primitive of the imprimitive sequences.

The first section describes the Phrase Process Sequence, which sets up a phrase in the IR and then calls the Pre-Operation sequence to process it. The Pre-Operation sequence, described in the second section performs the various pre-operations which may be specified by an Operand Phrase: pushing of the PR, modifying of the PR value field, etc.

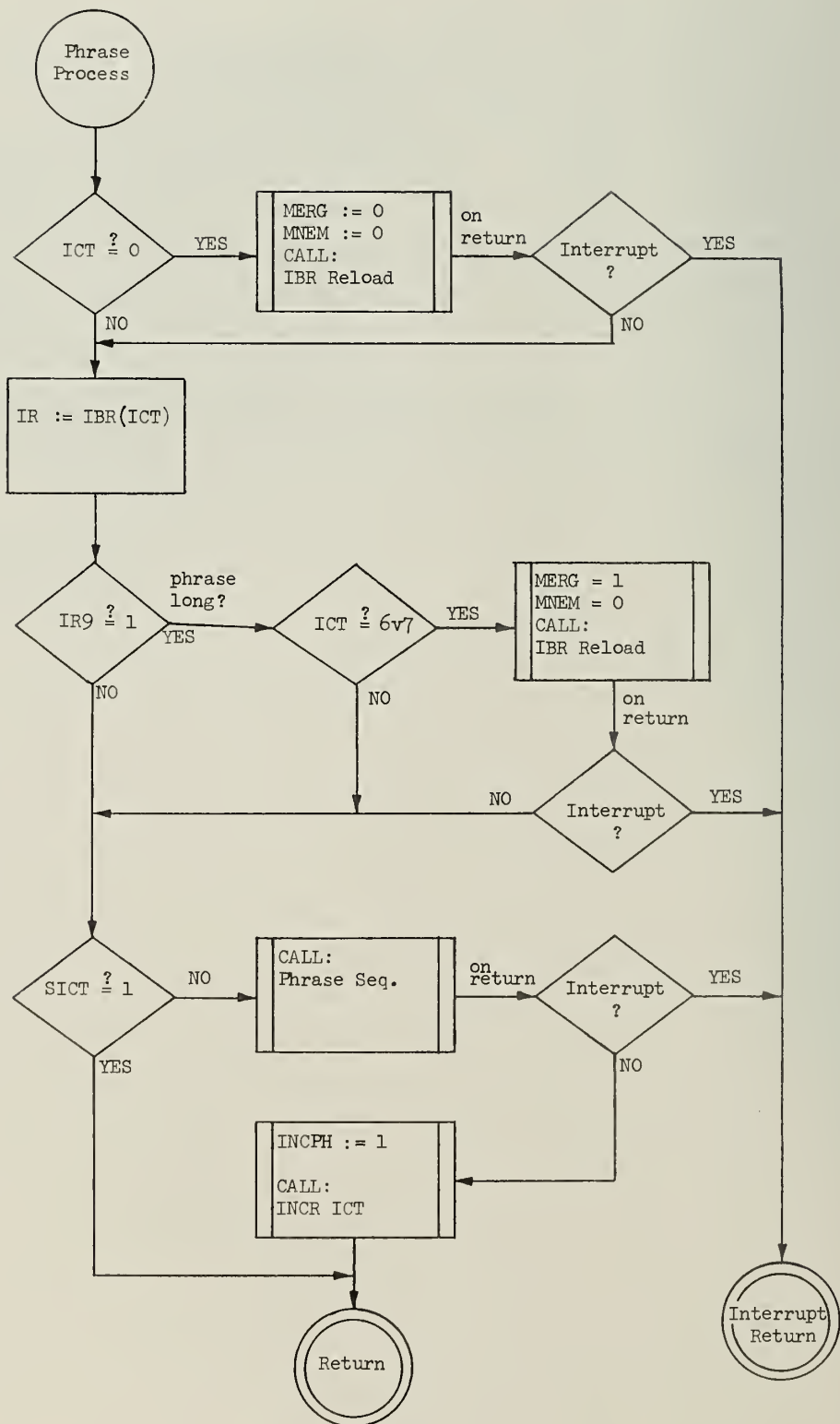
The remaining sections describe the Post-Operation Sequence, Increment ICT, which increments the instruction counter, IBR Reload and Exchange PR-SBR.

#### 4.4.1 Phrase Process Sequence

##### 4.4.1.1 Phrase Process Sequence Description

The Phrase Process Sequence is used to control the initial processing of a phrase. This includes checking the ICT for zero and reloading the IBR if necessary, loading the IR from the IBR, loading the tag register, TGR, from the leftmost byte of the IR and finally performing the phrase operations and ICT incrementation. In this sequence the phrase operation will not exchange PR#0 and the SBR if the TGR = 0.

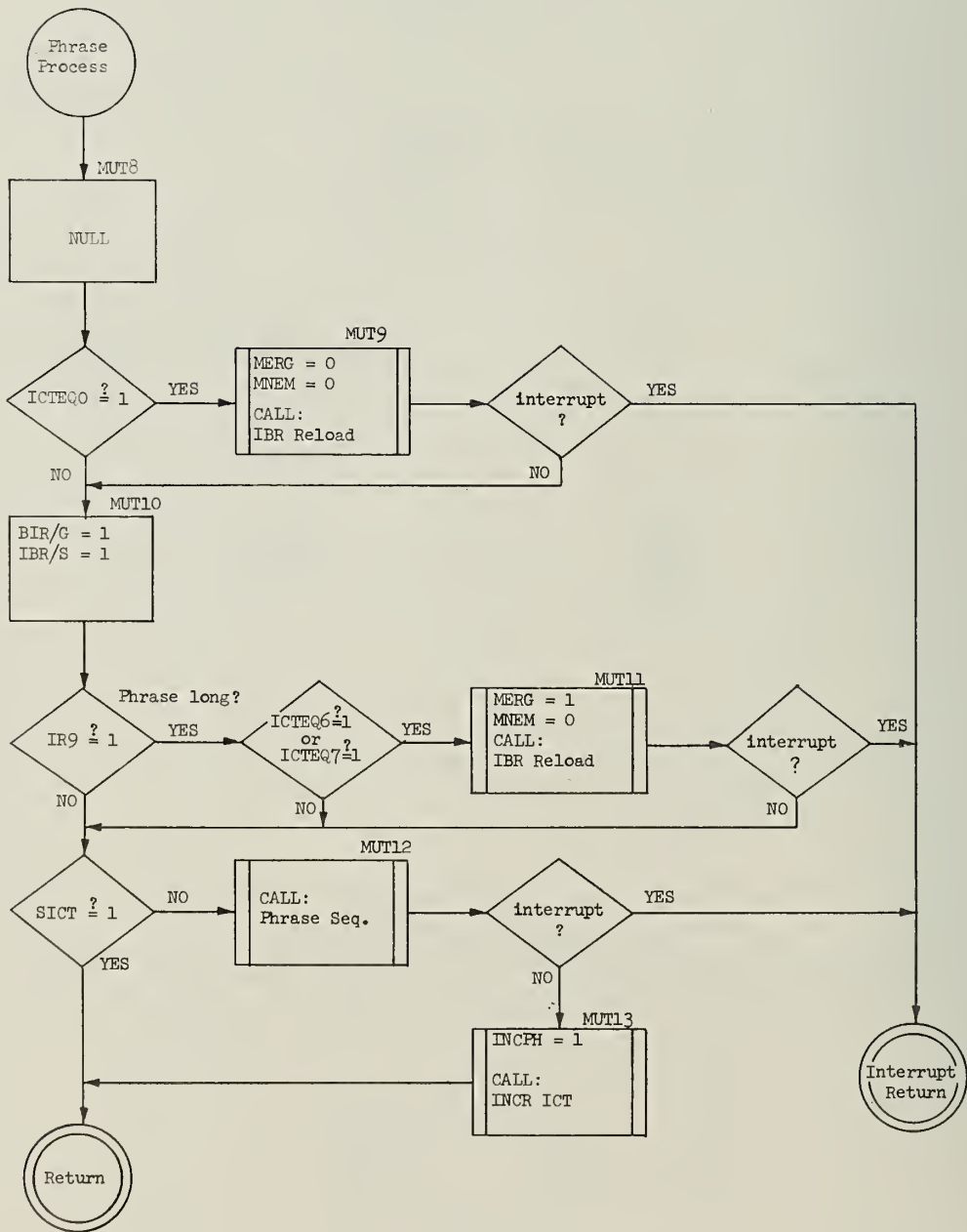
There is one option which is used in the imprimitive sequence which skips the phase operations and the ICT incrementation.



Phrase Processing Sequence Flow Chart

#### 4.4.1.2 PHRASE PROCESS Control Logic

As shown in the control step flow chart at the end of this section, the Phrase Process Control logic is very straight forward. There is one control signal, SICT, which, if on, causes the sequence to skip around the phrase operations and the incrementation of the ICT.



Phrase Processing Sequence - Control Step Flow Chart

#### 4.4.2 Phrase-Operation Sequence

##### 4.4.2.1 Phrase-Operation Sequence Description

The Pre-Operation Sequence is used to perform the various PR operations which may be specified by the operand phrases before the instruction is executed. There are two entry points: the Phrase Sequence entry point is used by the operand phrases of the Primitive and Imprimitive instructions, and the Operator Sequence entry point is used by the first operand phrase of the Imprimitive instructions. The Operator Sequence entry point does not check for the immediate option or the pre-slash since these are not allowed in the initial operand (operator) of imprimitive instruction phrases.

The sequence is set up in such a manner that it assumes the phrase to be processed is always left-justified in the IR. The first operation of the sequence is then to load the tag field from the left-most byte of the IR into the TGR.

Next if the tag register is 13 the OS must be cleared. This insures that if the operand stack is accessed by means of a PR reference all data in the hardware registers of the TP will have previously been put into core memory.

If the sequence was entered by way of the Operator Sequence entry point and if the phrase is short and not indirect the sequence is finished. If the sequence was entered by the Phrase Sequence entry point then the IMM flip-flop must be set provided that the instruction can have the immediate option and that the IR8 bit is 1. If no push of the PR is indicated by the phrase and if the phrase is short and not indirect then the sequence is finished.

If further operations are necessary then a check for TGR = 0 is made provided that the CKTGZ signal for the Phrase Process sequence has been set to one. Next if the Phrase Sequence entry was used and if IR5 is 1, then the PR stack sequence must be executed. If an interrupt occurs



in this sequence the STACK PR control logic will assure that the TP returns to its state prior to the call. Then in order for the PRE-OP sequence to return the TP to its state before the PRE-OP sequence was called, all that is necessary is to reexchange the PR and SBR if this had been done previously. Then an interrupt return can be performed.

If there was no interrupt and if the phrase is short and not indirect, then the sequence is finished. Otherwise it continues with both entry points using the same operations for the rest of the sequence.

Next, if the phrase is short, a single indirect address is performed and control of the sequence is given to the final check of the tag register for a tag of zero.

If the phrase is long and the indirect modifier option is being used, the tag register must be loaded with the secondary tag, i.e. the tag field from the second byte in the IR (bits 10 through 13). If the secondary tag is 15, the AR is then loaded from the OS, otherwise the AR is loaded with the value field of the designated PR. When no indirect modifier option is specified, the AR is loaded from the modifier field in the IR.

At this point the AR has been loaded, right-justified, with a two-byte modifier field. The next step is to determine what to do with this modifier.

If the modification option is addition the modifier is added to the PR value designated by the primary tag. If replacement is specified the modifier replaces the value.

If conditional subtraction is indicated the modifier is subtracted from the PR value and the appropriate actions are taken depending on the result.

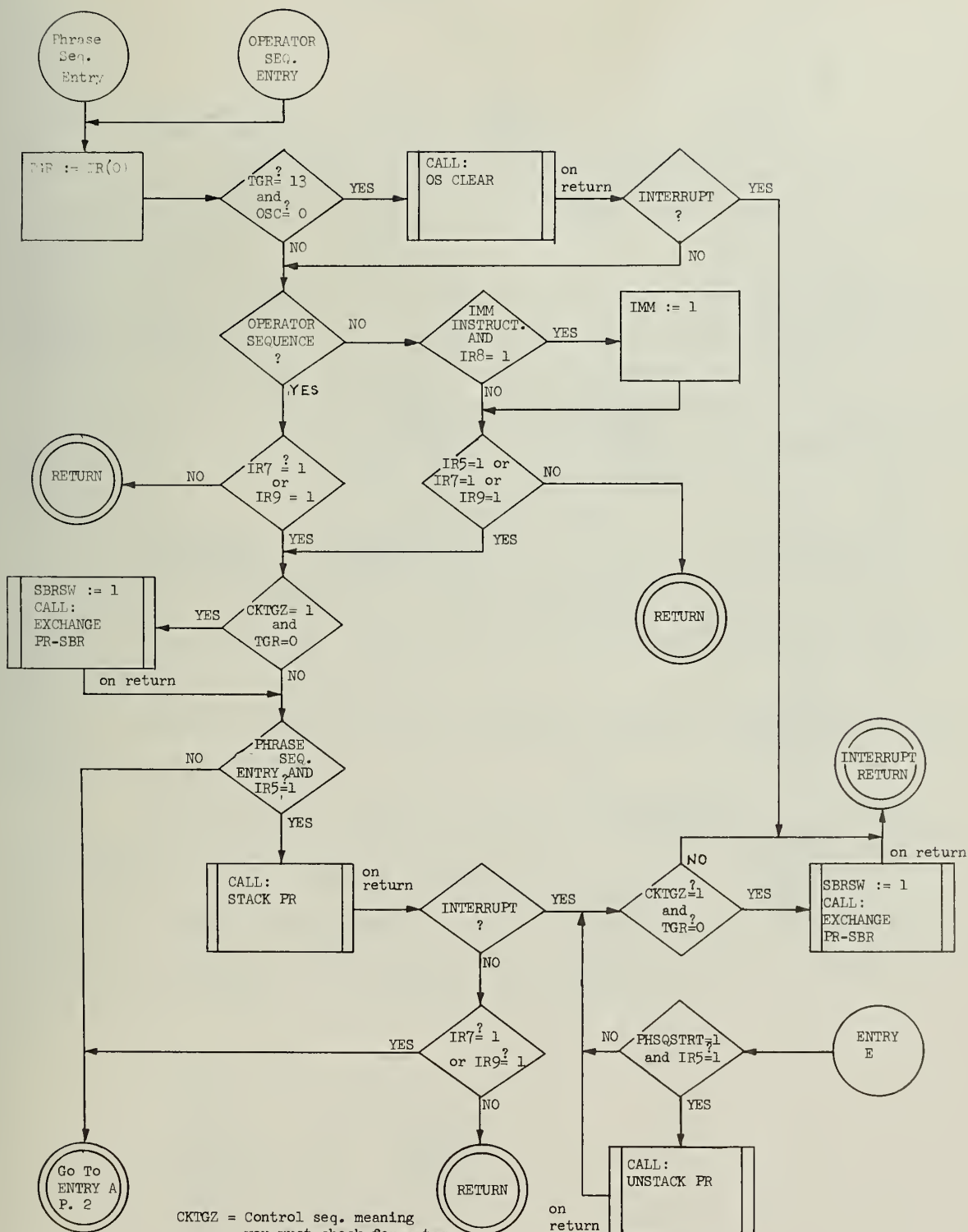
Finally if the PR has been modified during the sequence, the tag is checked. If the tag is zero, PROMOD must be set to one.

It should be noted that all interrupts which may occur as a result of this sequence can be "bucked upstairs", provided that certain assumptions can be met. In the logic, if an interrupt occurs after the PRE-OP sequence has stacked a PR, then in order to restore the TP to its original state the PR must be unstacked before the sequence can make an interrupt return. However in the most general case an interrupt could occur in the UNSTACK PR sequence; thus this operation could not ordinarily be performed after one interrupt has occurred since there is no provision for the control logic to handle stacked interrupts.

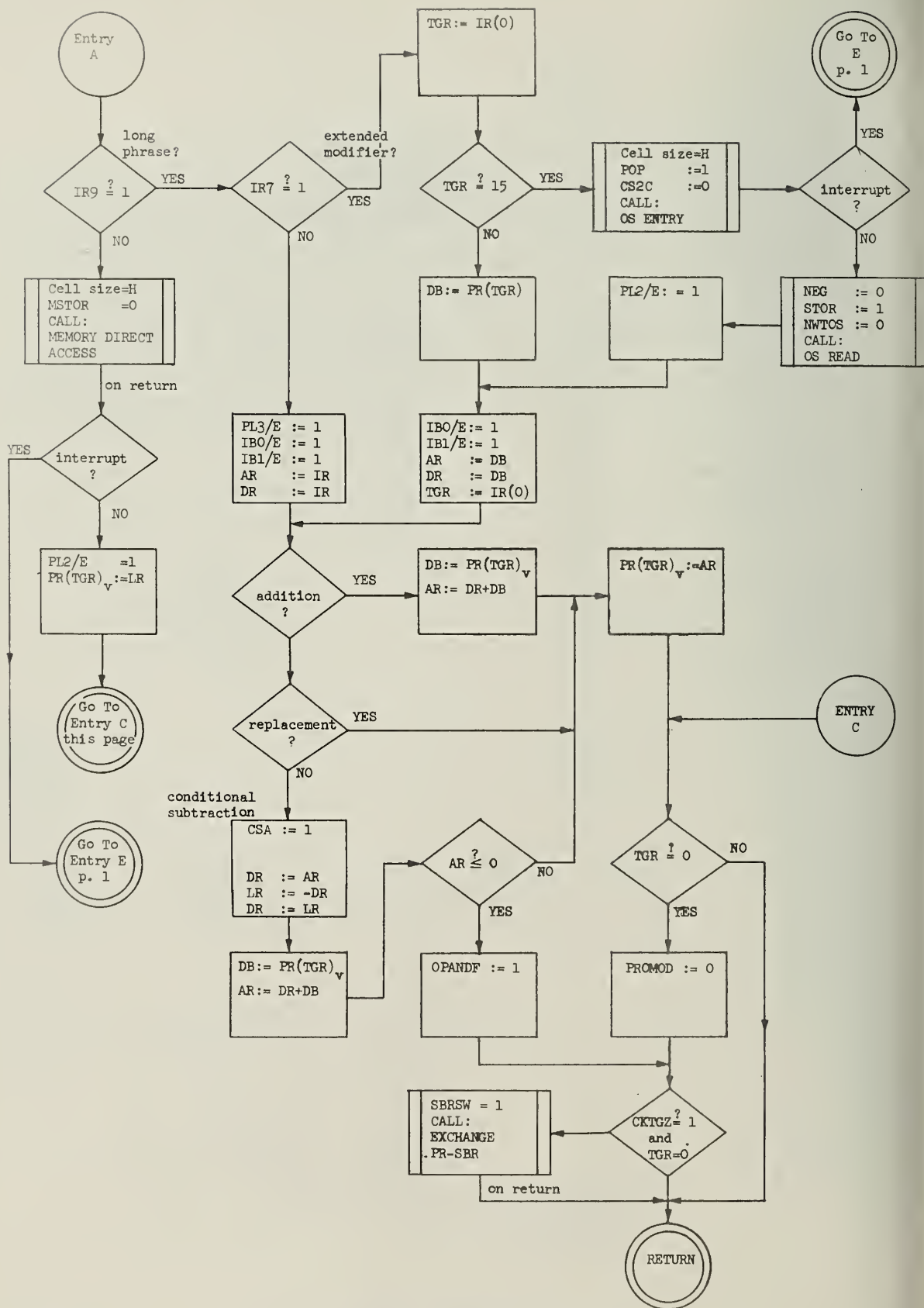
To circumvent this difficulty and to execute the sequence we have assumed that it will not be interrupted if the preceding STACK PR sequence was OK. This reasoning is based on the fact that UNSTACK PR cannot cause any interrupts except those concerning the accessing of the cell being unstacked and this is identical to the cell which was previously manipulated by STACK PR. Thus no interrupt will occur in the UNSTACK PR sequence if the following conditions are met:

- 1) The previous STACK PR sequence call did not generate an interrupt .
- 2) A cell which can be written into without being interrupted can be read without being interrupted.
- 3) No change of a System Name Table is possible while the program using it is active (i.e. running on a TP).
- 4) No hardware failure occurs in the memory box containing the unstacked cell between the time the STACK PR and UNSTACK PR sequences are executed ( several hundred nano-seconds).

Given the Operating System constraint detailed above, only this last occurrence is actually possible. However, its probability of occurrence should be exceedingly low. Therefore we shall go ahead and perform the UNSTACK PR sequence if necessary during an interrupt. If an interrupt does occur due to violation of assumption 4, two different interrupt conditions will be set when the Interrupt Control sequence is executed. This can be treated as a fatal interrupt for the program since a hardware memory error usually is just that.



Phrase Operation Sequence Flow Chart



Phase Operation Sequence Flow Chart

#### 4.4.2.2 Phrase Operation Control Logic

The PHRASE OPERATION sequence performs all of the necessary pre-operations on an instruction phrase. The control logic involved is pretty extensive.

MU37 is rather complicated since it can be entered from three different positions and each position causes a slightly different set of gates to be turned on. Note that IBCO, IBC1, BAR/G, and REQ/S are always turned on. The other signals are chosen as shown in Figure 4.4.2.2/1. Note that MU37 will have to remain on long enough for the longest operation to take place.

In order to negate the quantity in the AR for a conditional subtraction, control points MU40 and MU41 gate the AR into the DR, generate all 1's on the DB and perform an exclusive or into the LR. A carry is then injected in the low order bit during the addition in order to make the final result 2's complement.

Note that MU45 performs one of two functions depending on which control point activates it.

Note that MU37 leads directly to MU38. This is not strictly necessary when  $IR7 = 0$  since in this case the TGR was not loaded from the second byte in the IR (byte No. 1) and there is thus no need to reload it from the leftmost byte again (byte No. 0). However since MU38 will take relatively little time the logic necessary to skip around it seemed to be a waste.



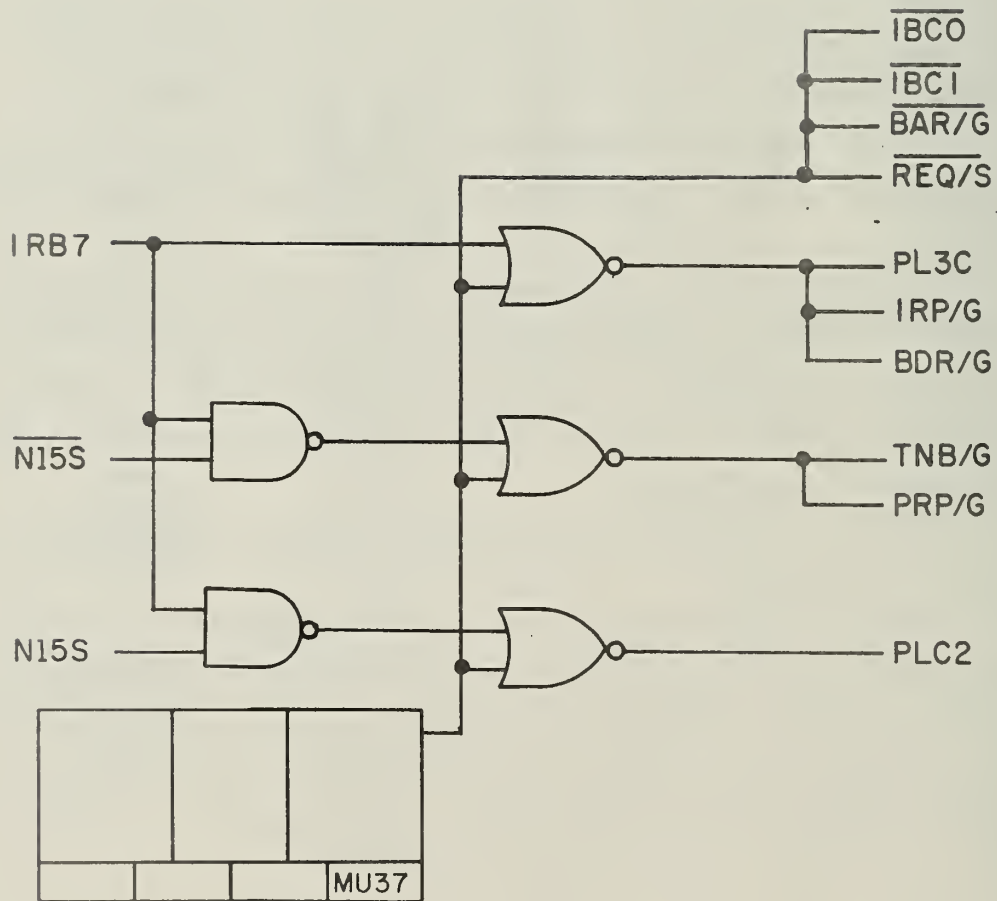
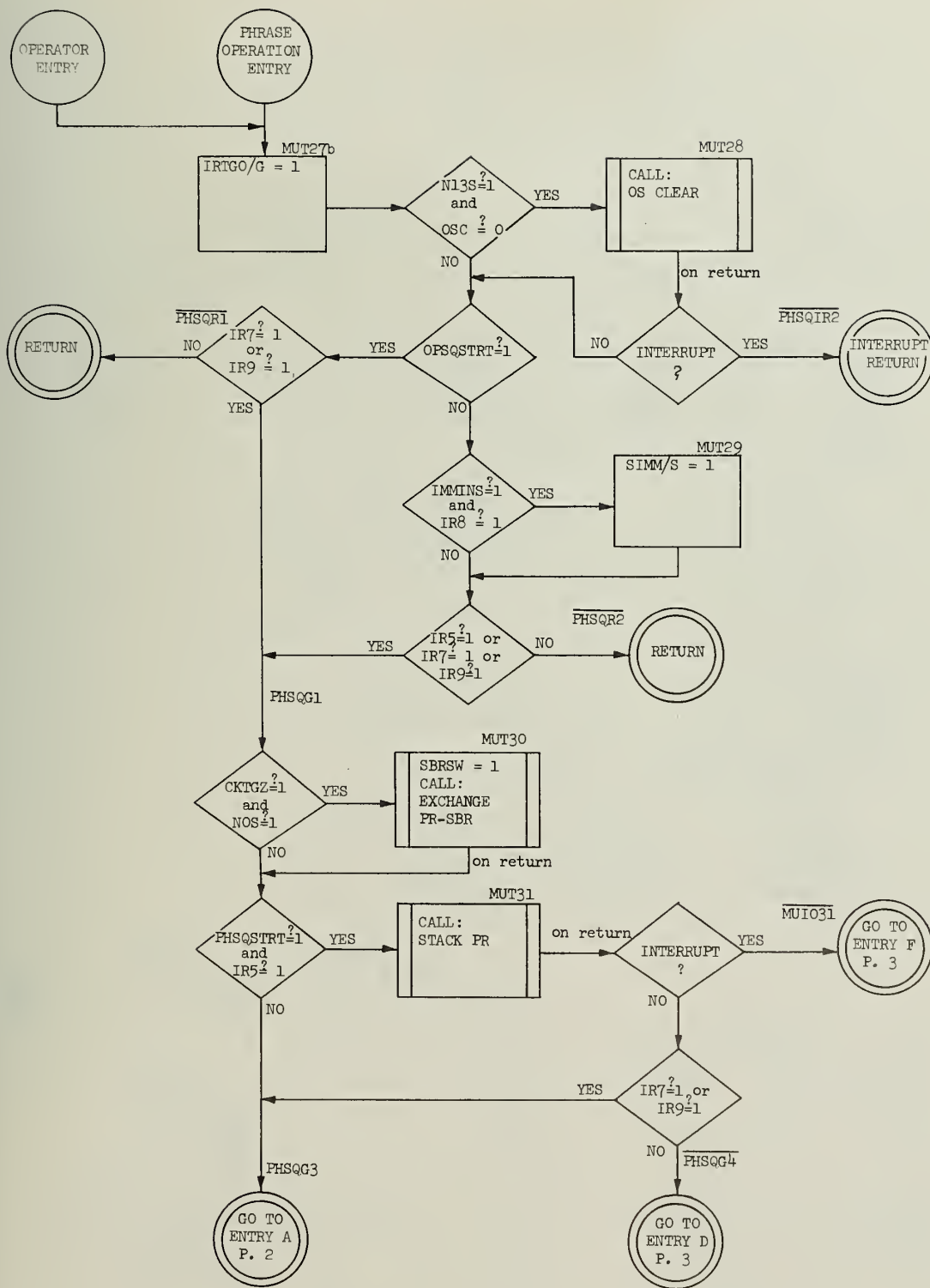
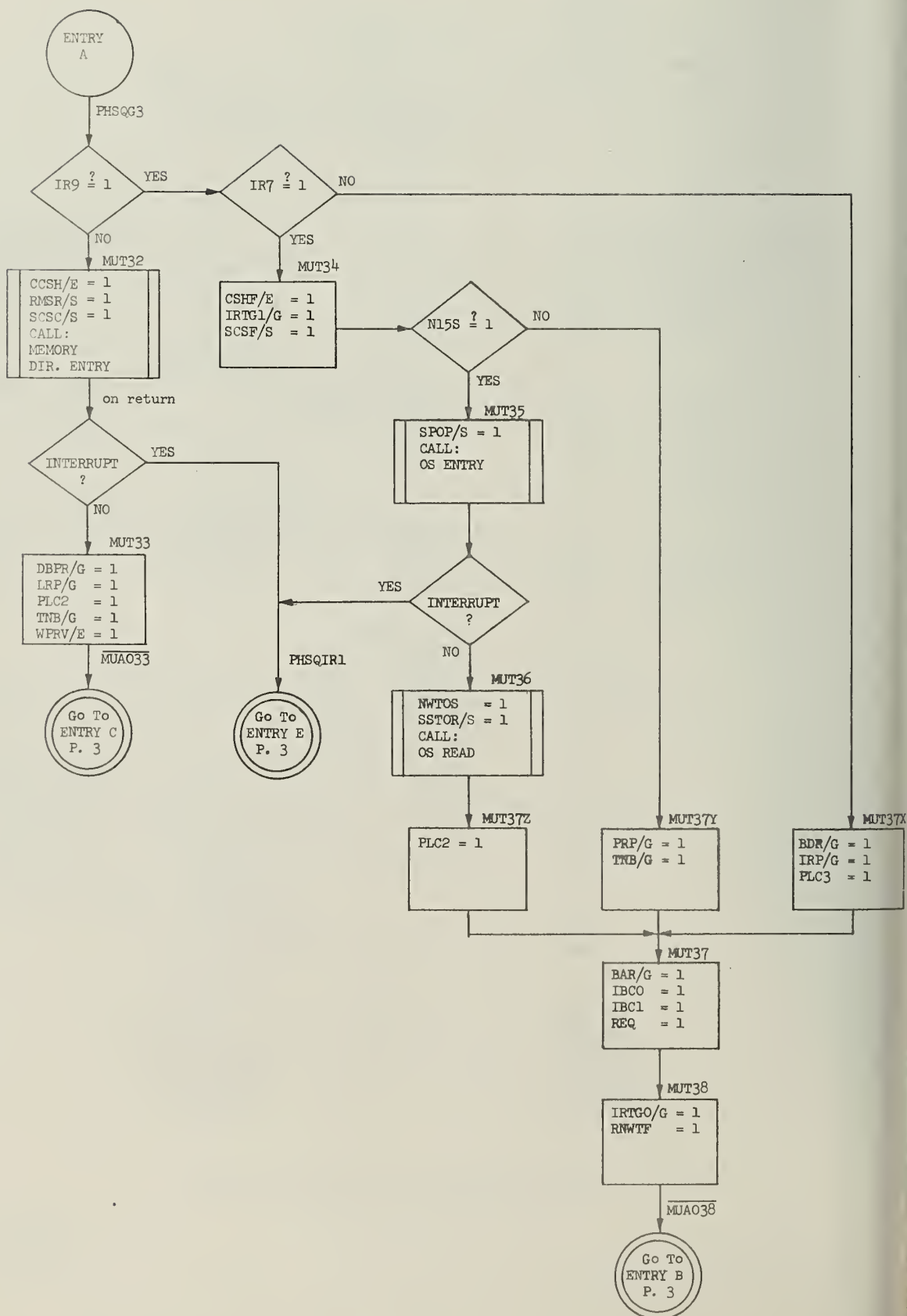


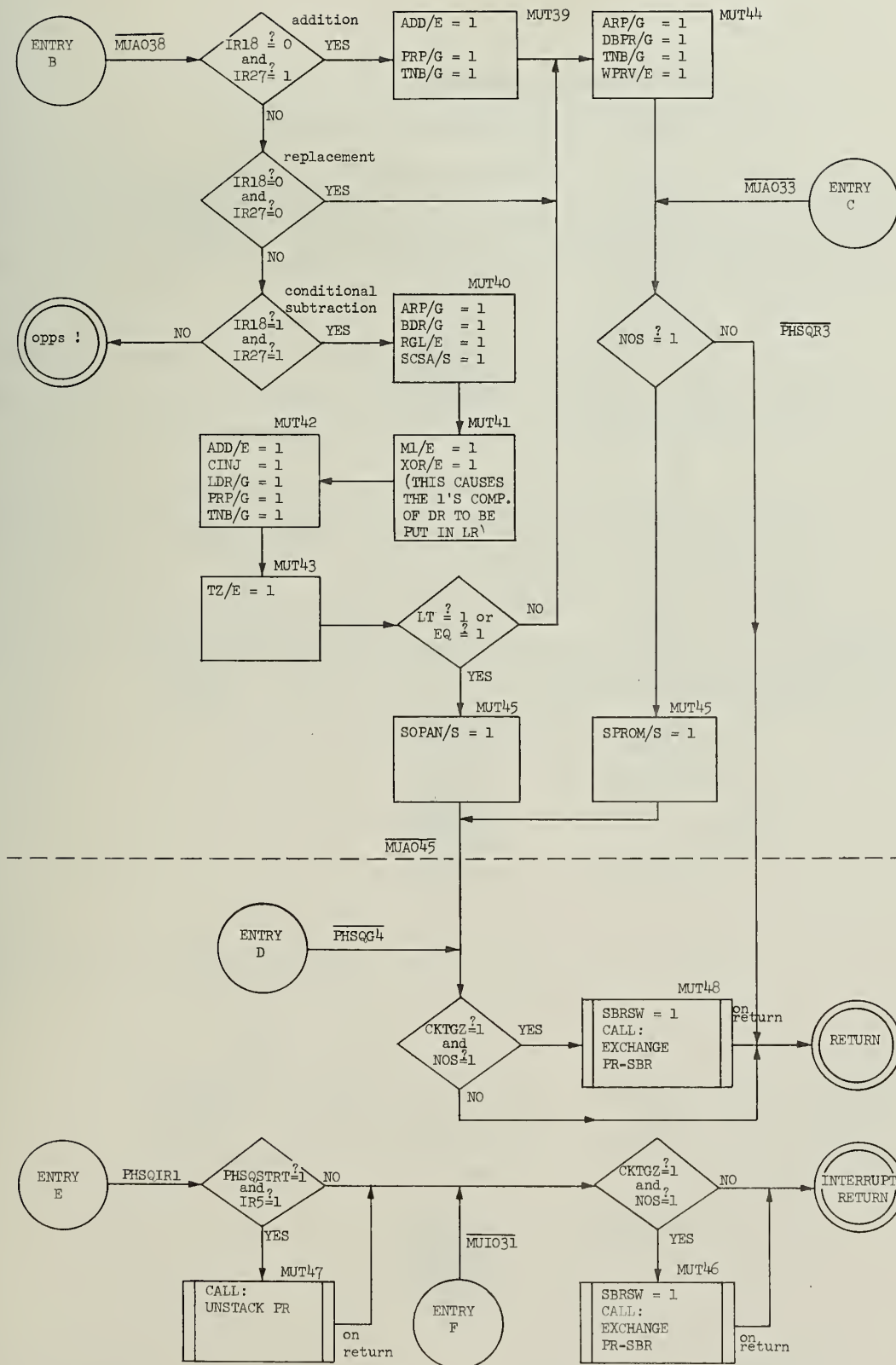
Figure 4.4.2.2/ 1 Task Signal Logic for MU37





Phrase Operation Sequence - Control Step Flow Chart





#### 4.4.3 Post-Operation Sequence

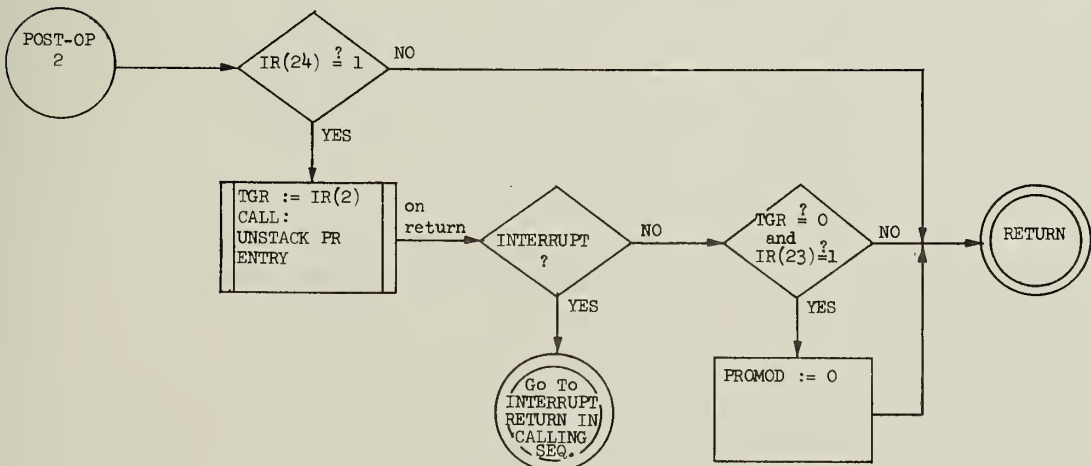
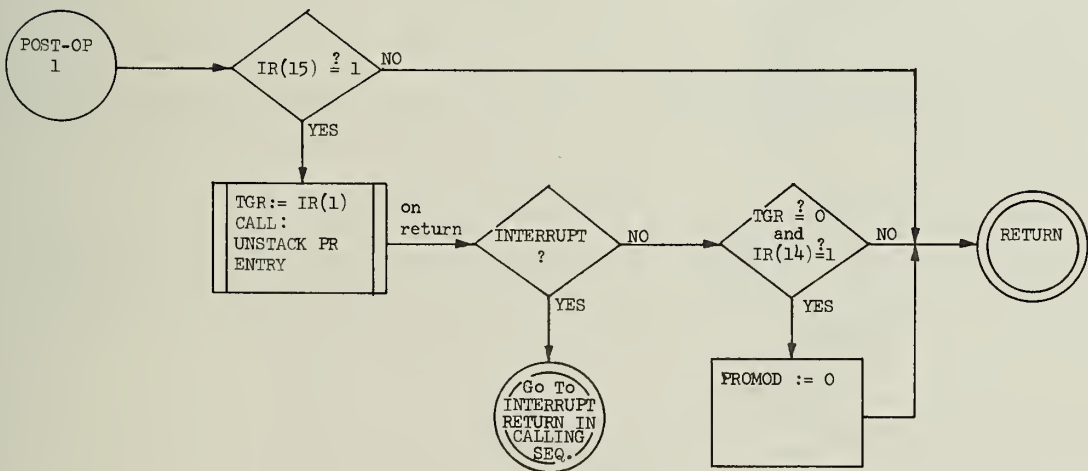
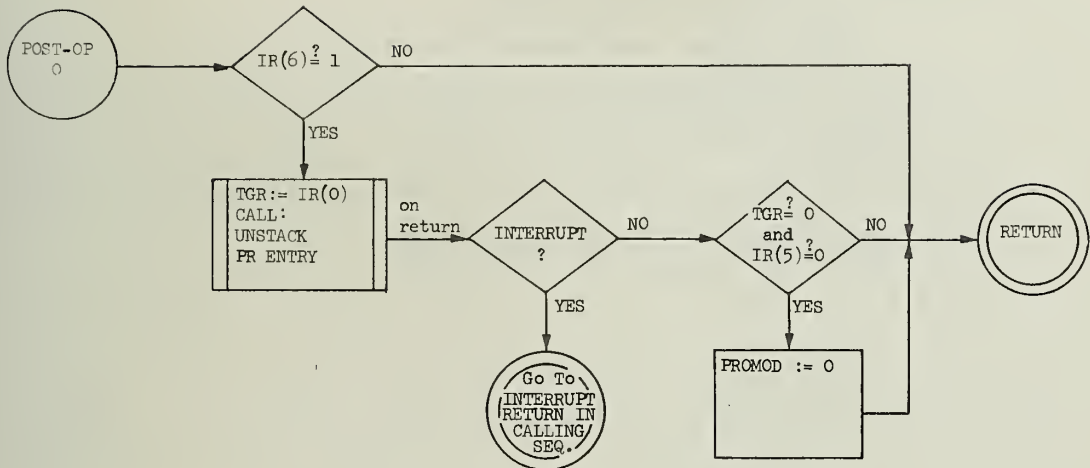
##### 4.4.3.1 Post-Operation Sequence Description

The Post-Operation Sequence is used to perform the post-slash operations as called by the Final Control sequence. There are in effect 3 separate entry points which perform the same operations but test different bytes in the IR.

The Post-Op 0 sequence performs its operations using the phrase data stored in the 0th byte of the IR. The Post-Op 1 and Post-Op 2 sequences use the 1 and 2 numbered bytes in the IR.

The operations consist of loading the TGR with the proper bits from the IR, and then testing the post-slash bits. If there is no post-slash the sequence returns. If there is a post-slash the sequence calls the Unstack PR sequence after which it tests for a TGR of 0. If the TGR = 0 and there are both pre-and-post-slashes, PROMOD is set to zero.

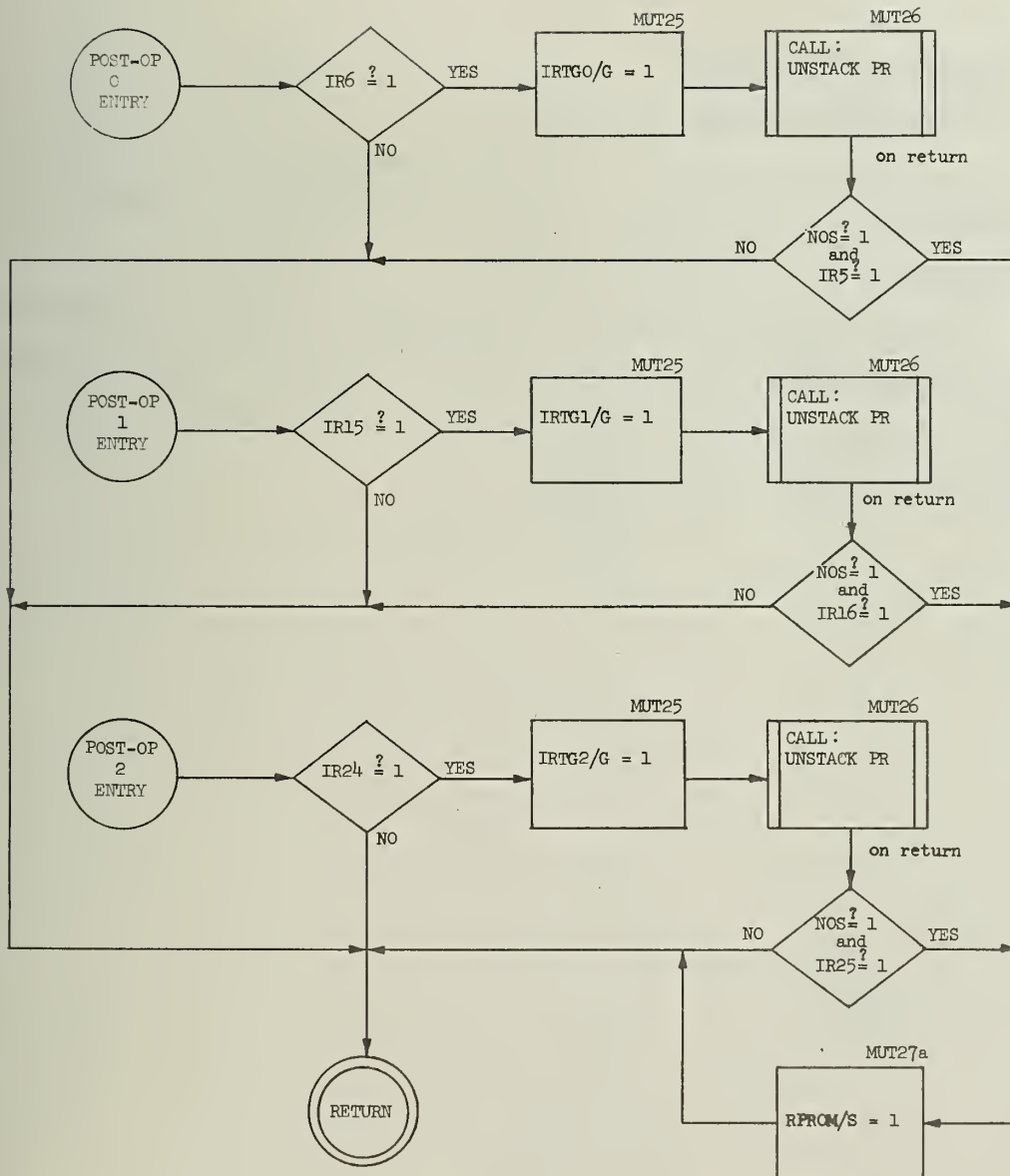
Then the sequence returns.



Post - Operation Sequence

#### 4.4.3.2 POST-OP Control Logic

The three versions of the POST-OP sequence are all implemented using the same control points. Note that since the POST-OP sequence may be called 3 times in a row in the Final Control Sequence, care must be taken when designing the logic so that the sequence will have time to reset itself before being called again.



Post - Operation Sequence - Control Step Flow Chart

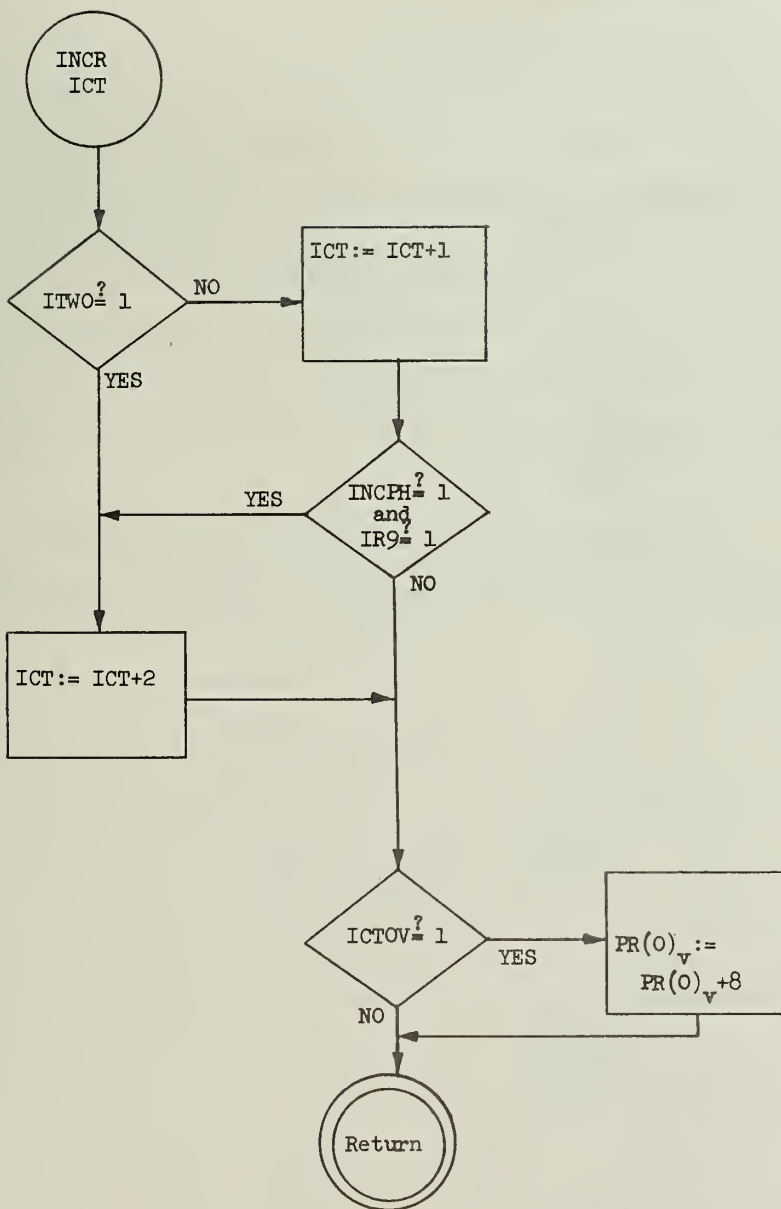


#### 4.4.4 Increment ICT Sequence

##### 4.4.4.1 Increment ICT Sequence Description

The INCR ICT sequence is used to increment the ICT by +1 or +2 and then to check for overflow. If there is an overflow, PR#0 must be incremented by 8.

Another option is available in which the ICT will be incremented past the current operand phrase whose contents are being held left-justified in the IR. In this situation the ICT will be incremented by 1 if the phrase is short and 3 if the phrase is long.



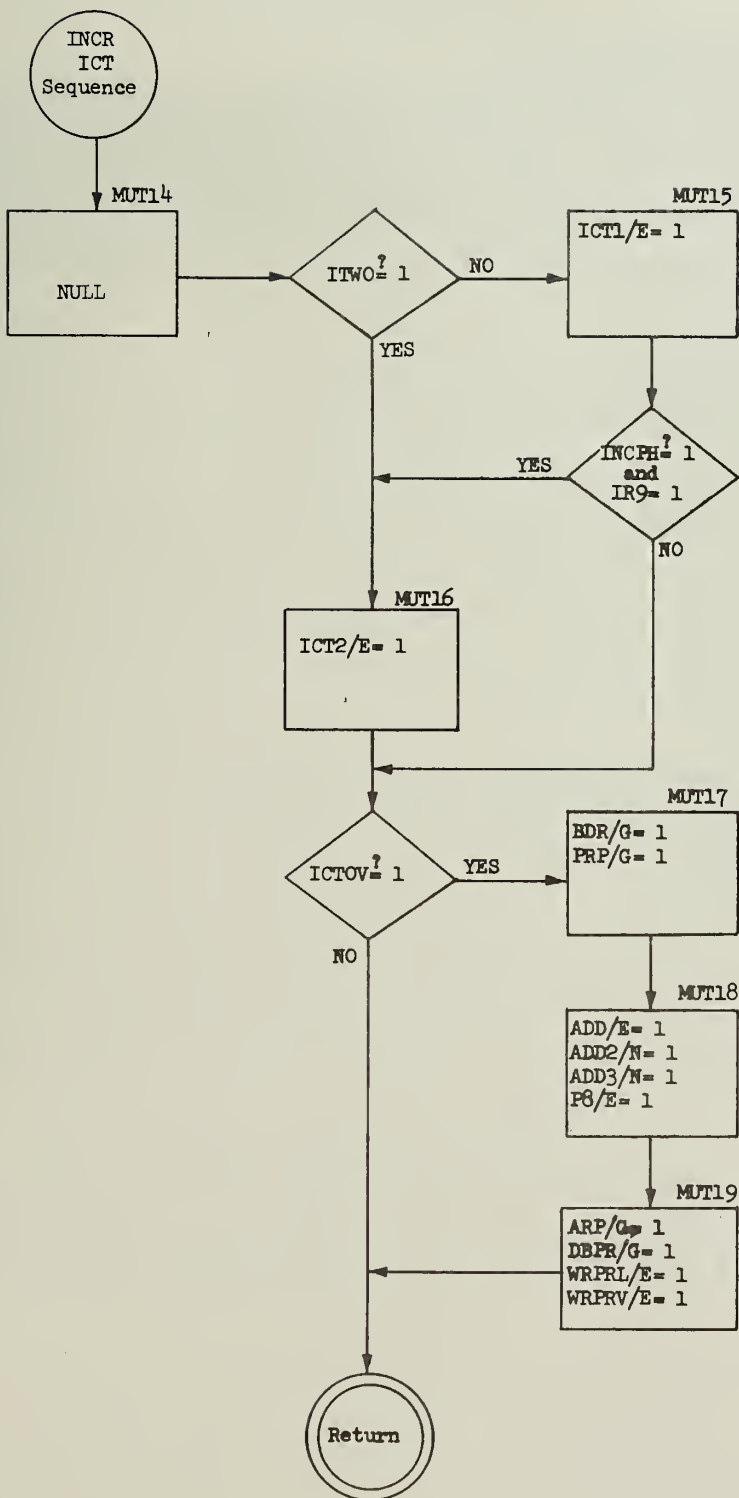
INCPH = 1 - increment ICT over phrase

ITWO = 1 - increment ICT by 2

#### 4.4.4.2 INCR ICT Control Logic

The INCR ICT sequence is entered via a null control point so as to allow the initiating signals to change state without adversely affecting the sequence. Next, the signal ITWO is used to determine whether there are two operands or **not**. If two, ICT2/E is turned on, incrementing the ICT by 2 bytes. If not, ICT1/E is turned on, incrementing the ICT by 1 byte. Then the signal INCPH and IR9 are directed to determine whether the ICT should be incremented past the phrase (INCPH = 1), and whether the phrase is long or short (IR9). For a long phrase, the ICT is incremented by 2, resulting in a total increment of 3 bytes.

The signal ICTOV indicates an ICT overflow. If an overflow has occurred, control points MUT17-19 are used to increment PR#0 by 8.



Increment ICT Control Step Flow Chart

#### 4.4.5 IBR Reload Sequence

The IBR Reload Sequence is used to reload the IBR whenever it is emptied and the complete instruction has still not been obtained. It is used by all of the Main Control Subsequences: Primitive and Imprimitive.

#### 4.4.5.1 IBR Reload Sequence Description

The IBR Reload Sequence consists of two parts: loading the Instruction Buffer Register with the next double word in the instruction stream and merging the new data into the IR. The first half is accomplished by adding 8 to the contents of PR#0 and making a double-word read access to memory. The second part requires quite a bit more explanation.

As was mentioned in Section 2.6.1.2, if the ICT is greater than 4 at the beginning of an instruction fetch, a "wrap-around" will occur when the IR is loaded from the IBR. Thus one or more of the initial bytes of the IBR will be loaded into the IR following the valid instruction bytes. If, subsequently to this, the IBR is reloaded, it will be necessary to reload these "invalid" instruction bytes in the IR with the correct ones which were just stored in the IBR. However we must not destroy the valid instruction bytes in the IR since these particular bytes are no longer in the IBR.

The way to do this involves a merging circuit connected to the control logic of the Permuter. There are several considerations. First, if there is a mnemonic byte in the IR (this will be true in all imprimitive and non-PAU primitive instructions and also in the first access for PAU instructions), the first byte of good data will be in the rightmost byte position. Second of all, the logic must set the gate inhibit signals of the IR.

The permutations are easily taken care of by the Permutation Control Logic (See Mainframe Logic Book Drawing 04-7). If a mnemonic byte is present OPJ/E is turned on. Thus the merge logic need only take care of the inhibiting signals as shown in Figure 4.4.5.1/1.

For non-mnemonic byte reloadings:

if ICT =	Gate Inhibits
1 1 1	0,
1 1 0	0, 1,
1 0 1	0, 1, 2,

For mnemonic byte reloadings:

if ICT =	Gate Inhibits
1 1 1	3,
1 1 0	3, 0,
1 0 1	3, 0, 1,

Figure 4.4.5.1/1 - Signals Used In IR Merging



The logic necessary for the merge circuit control is shown in Figure 4.4.5.1/2. Note that there are two merge signals. The one which is used depends on the type of merge. The logic equations for this circuit are as follows:

$$\text{BIR0/N} = \text{MGIM} \vee \text{MGPM} \cdot (\overline{\text{ICT2}} \vee \overline{\text{ICT3}})$$

$$\text{BIR1/N} = \text{MGIM} \cdot (\overline{\text{ICT2}} \vee \overline{\text{ICT3}}) \vee \text{MGPM} \cdot \overline{\text{ICT2}} \cdot \text{ICT3}$$

$$\text{BIR2/N} = \text{MGIM} \cdot \overline{\text{ICT2}} \cdot \text{ICT3}$$

$$\text{BIR3/N} = \text{MGPM}$$

These equations are implemented using a -236 diode matrix board or the IC chip logic shown in Figure 4.4.5.1/4. A flowchart showing the operations necessary to reload the IBR and merge it into the IR is given at the end of this section.

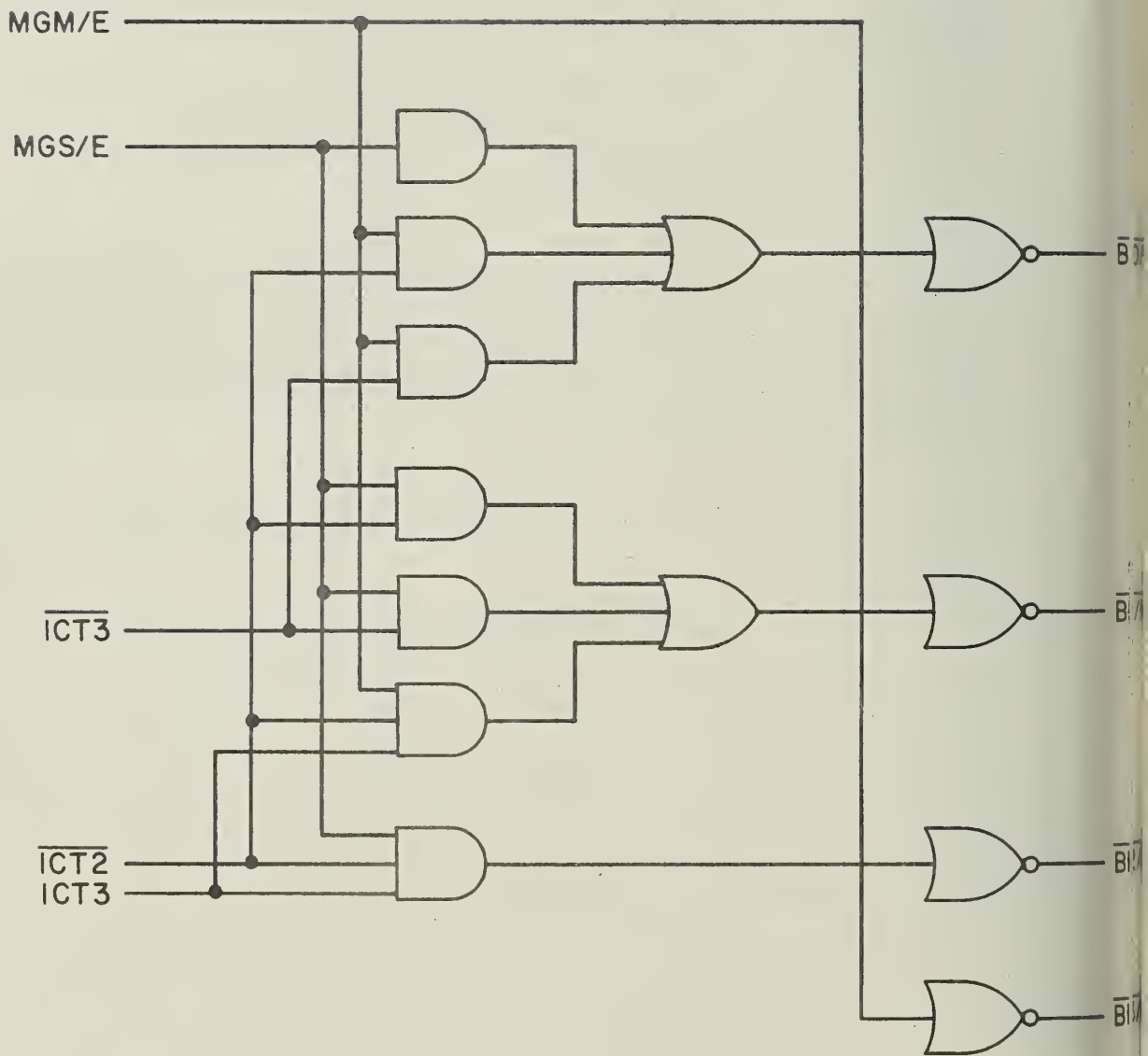
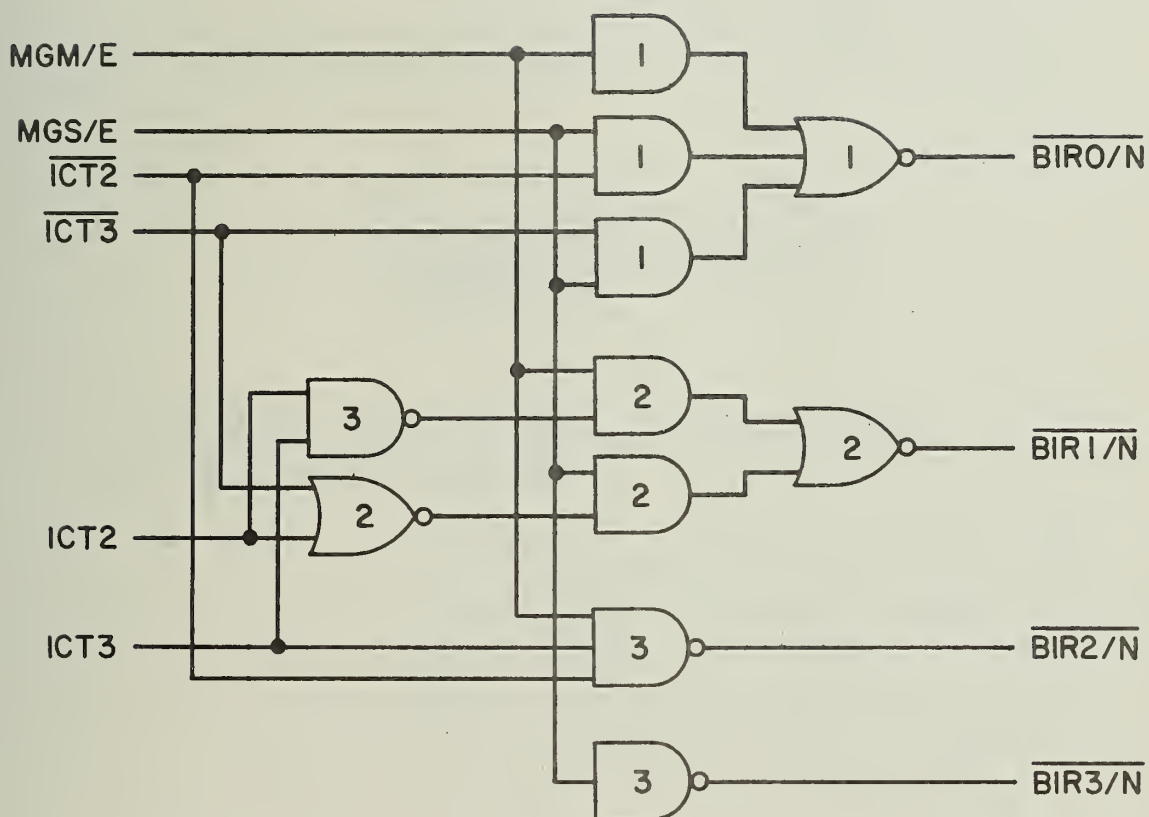


Figure 4.4.5.1/2  
IR Merge Control Logic

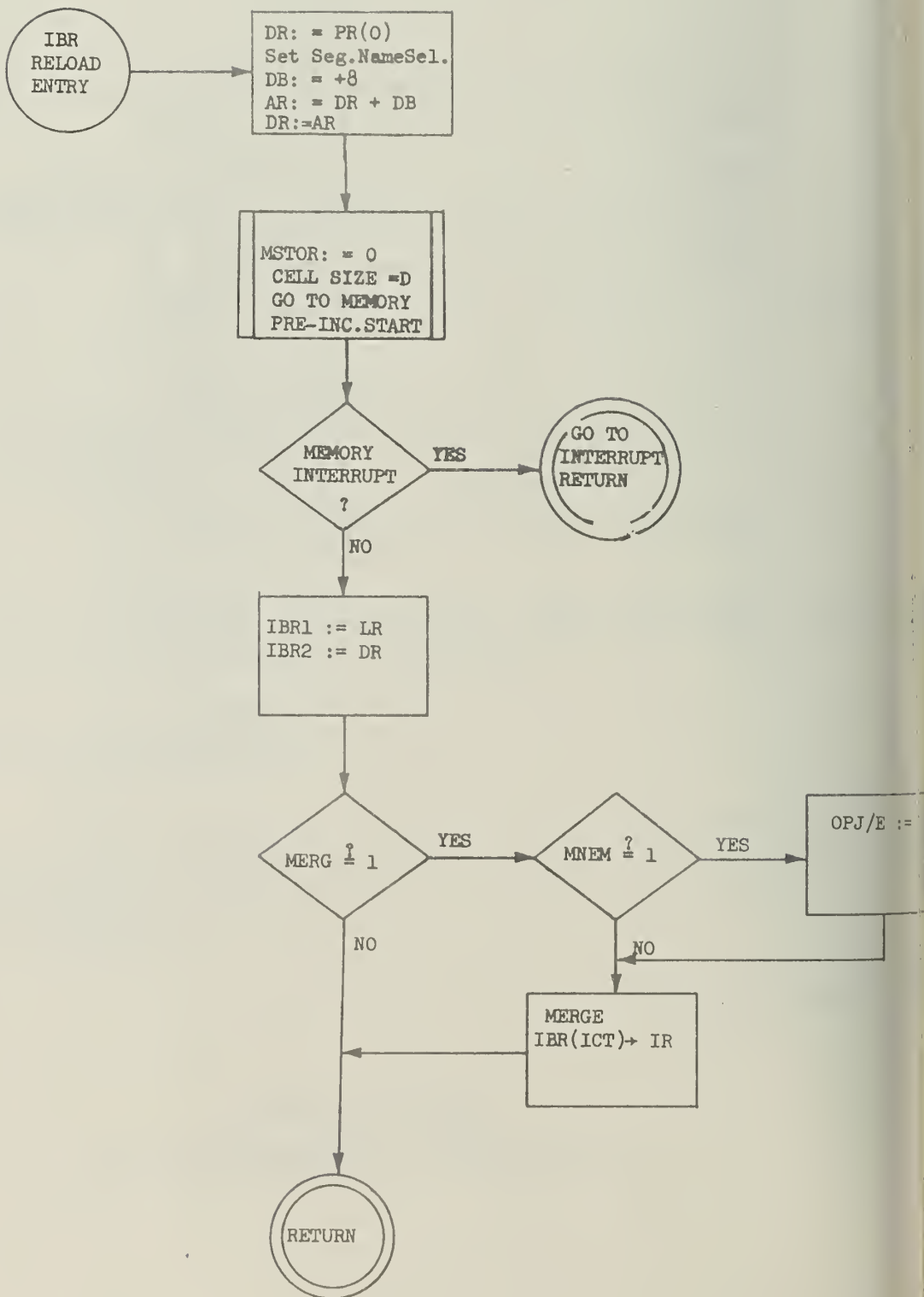


uses 3 IC chips

time: 1 collector delay after  
gate signal

Figure 4.4.5.1/3

IC Realization of IR Merge Control Logic



IBR Reload Sequence

#### 4.4.5.2 IBR RELOAD Control Logic

The IBR RELOAD ENTRY control logic makes use of two control signals: MERG which is a 1 if after the IBR is loaded a merge into the IR is desired, and MNEM which is a 1 if the IR contains a mnemonic byte and must therefore be loaded into the IR as shown in Figure 4.4.5.2.

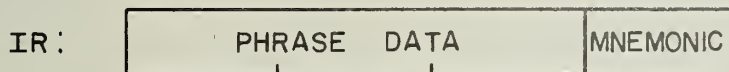
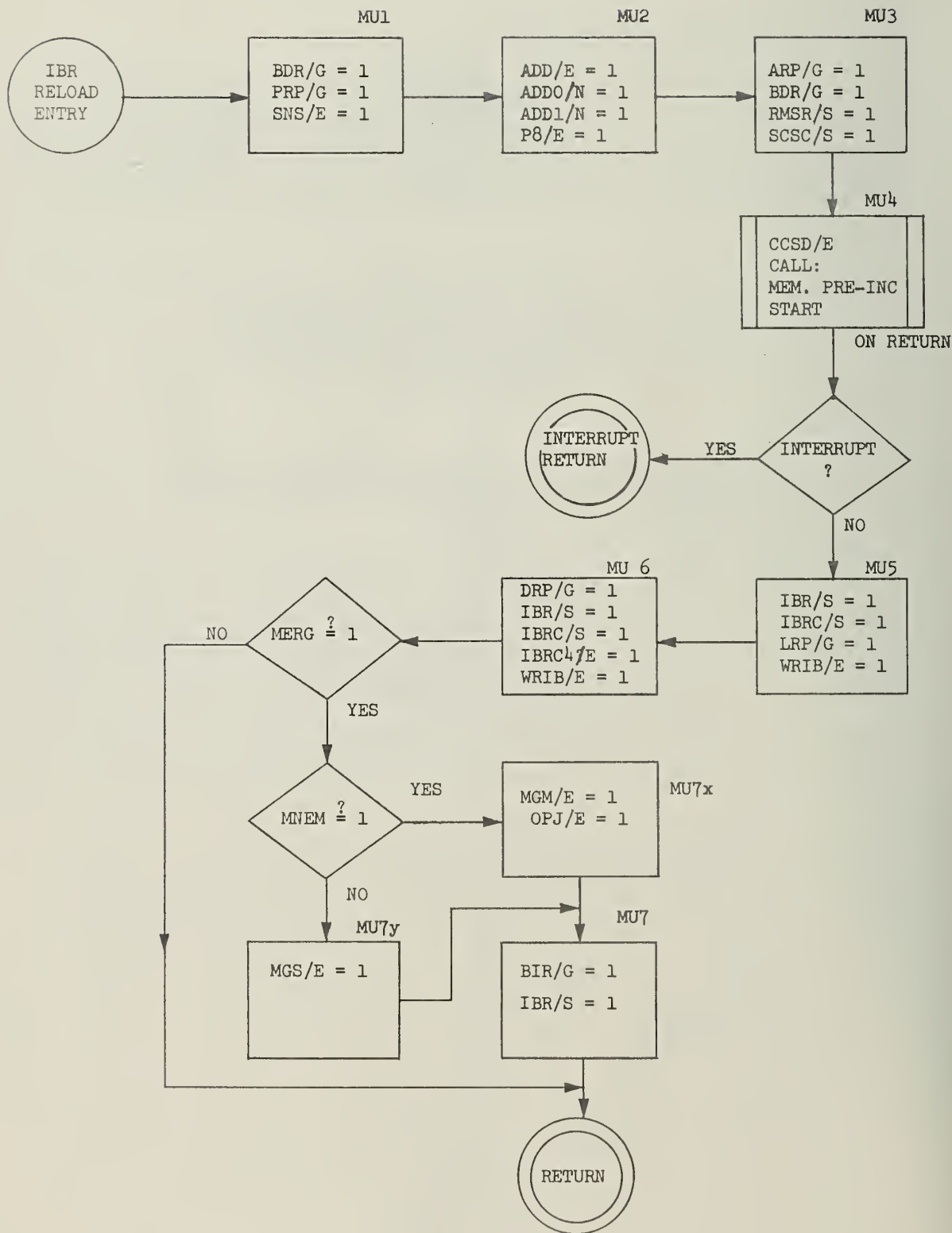


Figure 4.4.5.2 IR Format for Mnemonic Bytes

There are no significant complications in the control point logic. However it should be noted that in performing the memory access the cell size selector is switched to the control select option for double words. This means that the original setting is lost. In most cases this is unimportant since the IBR RELOAD sequence is usually executed before the cell size option for the instruction has been selected.



TP Main Utility Sequences  
IBR Reload Control Step Flow Chart

#### 4.4.6 Exchange PR-SBR Sequence

##### 4.4.6.1 Exchange PR-SBR Sequence Description

The Exchange PR-SBR Sequence allows for an exchange of value fields between two registers of the TP (usually a PR and the SBR). This sequence is used fairly often in the imprimitive sequence since the SBR is used as a temporary storage register for various PR values.

In general the sequence loads the DR with a pointer register and the LR with the SBR. The PR can either be PR#0 or be determined by the TGR.

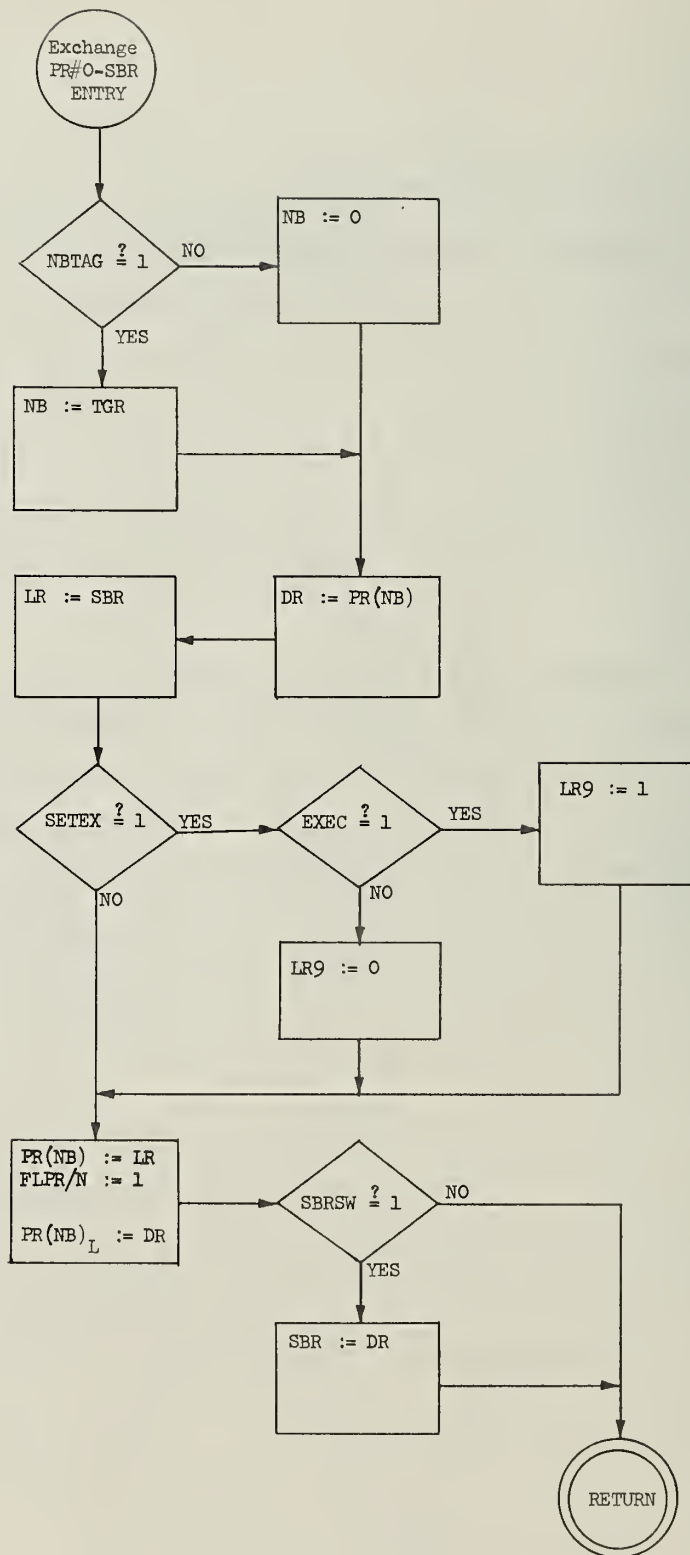
After the DR and LR are loaded from their chosen storage registers, the LR is stored into either the PR indicated by the TGR or PR#0. Then the link portion of the DR is loaded into the same register. This in effect results in only an exchange of value fields and flags while the link field remains unchanged.

If it is desired to also load the SBR, then the DR is stored in the SBR. There is no need to worry about the link field of the SBR since it does not contain anything valuable during an imprimitive instruction.

There is one other option allowed in this sequence. If desired, bit 9 of the LR can be set according to the state of the EXEC flip flop before the LR is loaded into the PR. This in effect will set the execute flip flop of the pointer register.

The flowchart for this sequence follows at the end of this section.





Exchange PR-SBR Sequence Flow Chart

#### 4.4.6.2 Exchange PR-SBR Control Logic

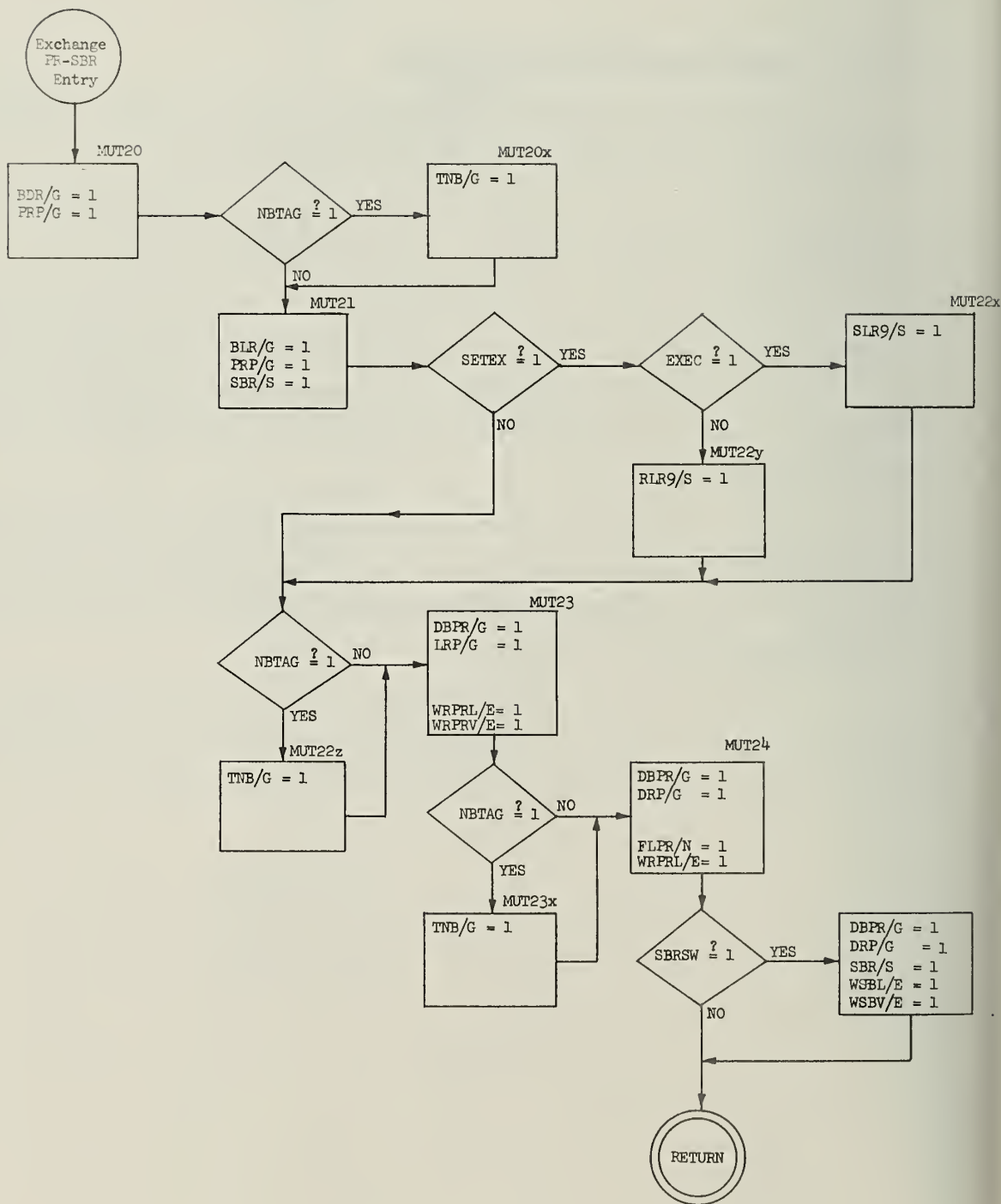
The Exchange PR-SBR Control logic uses four control signals which are set if necessary by the calling control point.

NBTAG is set to 1 if the pointer register being switched is to be designated by the tag register. Otherwise PR#0 is used.

SETEX is set to 1 if the execute flag of the PR in the exchange of data is to be set according to whether or not the EXEC flip flop is on, i.e. whether or not an EXECUTE instruction is being performed. If SETEX is 0, the value of the execute flag will be exchanged like all the other data.

SBRSW is set to 1 if it is necessary to load the data from the PR into the SBR. In some cases at the end of the imprimitive sequence there is no need to reload the SBR since the data will never be used. In these cases SBRSW is left off.

In control point MNT32 it should be noted that the RLR9/S and SLR9/S signals will force LR9 and  $\overline{\text{LR9}}$  respectively to zero, which in turn will cause the LR9 bit to stay in the desired state.



Exchange PR-SBR Sequence - Control Step Flow Chart

#### 4.5 Operand Stack Operations

The Operand Stack Operations are those processes which are common to all instructions and sequences which use the Operand Stack. These operations are broken down into two groups: the basic OS sequences which perform the operations directly needed to read from and write into the Operand Stack, and the supplementary OS sequences which perform clearing and initialization operations.

#### 4.5.1 Basic OS Sequences

##### 4.5.1.1 Basic OS Sequence Descriptions

The Operand Stack is controlled through the use of several basic OS sequences: the OS ENTRY sequence, the SCR MOD sequence, the OS READ sequence and the OS WRITE sequence. These sequences directly control the manipulation of the OS hardware and are used by all control sequences which must access the OS. The flowcharts describing these sequences are given at the end of this section.

The purpose of the OS ENTRY sequence is to check for potential overflow or underflow in the Operand Stack. The sequence makes sure that the portion of the Operand Stack which will be utilized in the execution of the instruction is available in the OS fast registers in the TP. If overflow or underflow is going to occur, the sequence makes the proper number of memory accesses to unload or fill the Operand Stack so that sufficient storage space is available for the operands. To determine how many bytes are needed in the stack, this sequence must know the type of operation (push or pop), the cell size (byte, halfword, word or double word) and the number of cells used (one or two).

The sequence begins by checking the type of instruction. If the instruction is a "pop" type a check must be made for underflow; alternatively if it is a "push" type an overflow check is made. In certain cases, i.e., the DUP instruction, it may be necessary to both read from the present stack and write additional data into the stack. This necessitates both an underflow and an overflow check.

The checks themselves involve using the overflow/underflow logic described in Section 2.3.1.4. The signals generated by this logic are valid as soon as the logic settles down after a change in the OSTR or SCR, i.e., the logic does not have to be "turned on". If an

overflow or underflow condition is detected it is corrected by making a memory access using the OSTR and either loading or unloading the proper OS fast registers.

There is one slight problem in using the OSTR. In order to get the full 16-bit virtual address of the double word in core which must be accessed, the OSTR must be masked into the low order 5 bits of the value of PR#13 (the lowest 3 bits are set to 0). However if the SCR has wrapped around the 31-0 byte boundary in the OS fast register storage, PR#13 will have been incremented by 32 and the wrong address will be obtained (see Figure 4.5.1.1). To correct this, the flip-flop WRAP is set to 1 every time the SCR and OSTR are pointing to different sides of the 31-0 byte boundary. The state of WRAP can be controlled by adjusting it each time either the SCR or the OSTR crosses the boundary. Thus when the OSTR is used in constructing an address, if WRAP = 1, then 32 is subtracted from the address before it is sent to the Memory Access Sequence.

As will be seen later the overflow correction part of this sequence is also used by the OS CLEAR Sequence. When this happens a special test is made before the OSTR is incremented. If the OSTR and the high order bits of the SCR are equal (and the OS not full) a special return is made to the OS CLEAR sequence without incrementing the OSTR. After the check has been made (and corrected if necessary) control returns to the calling sequence.

The SCR MOD Sequence is used to increment (SUB = 0) or decrement (SUB = 1) the SCR. This is done using the constant generator which generates a value equal to one or two times the cell size (depending on the number of cells which are needed). If the instruction is a "push" type this value is positive; if it is a "pop" the value is negative.



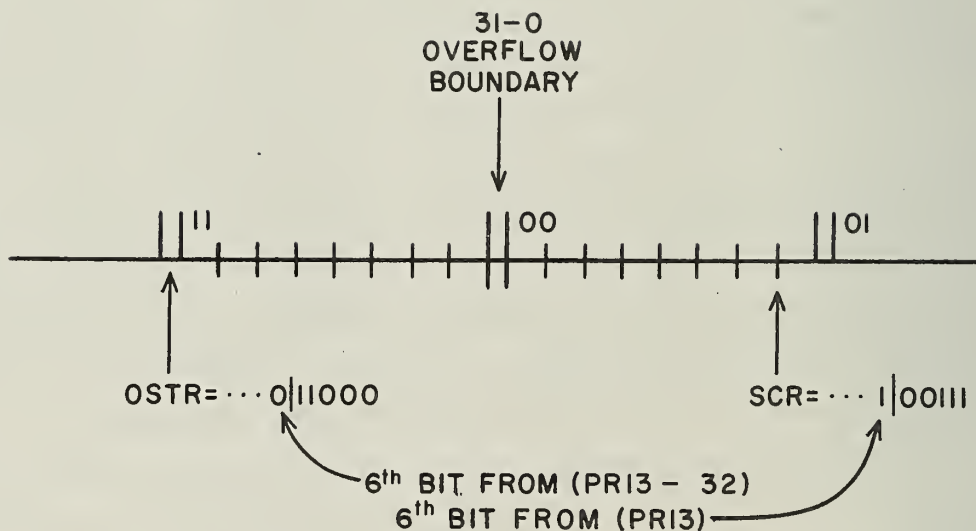


Figure 4.5.1.1 - SCR Overflow and Its Effect on the OSTR and PR#13

If the SCR and the OSTR points to different sides of the 31-0 boundary, their virtual addresses will differ by 32. However, high order bits (from the 6th least significant bit on up) are not returned by the SCR and the OSTR registers. By definition PR#13 reflects the value of the SCR. Accordingly whenever the address of the OSTR is desired, PR#13 will have to be adjusted by -32.



This value is then added to the current value of the SCR, using the 5-bit Adder. If there is an overflow on addition, or underflow upon subtraction, the A50V flip-flop will be turned on -- indicating that the SCR has passed the 31-0 byte boundary in the OS. Then if the change is to be permanent and if it is alright to destroy the contents of the AR, DR and LR the value of PR#13 is changed by +32. When this happens WRAP is set/reset according to the "direction in which the SCR is moving". If PR#13 must not be incremented/decremented at this time, the DECPR flip-flop is set to indicate that it should be adjusted as soon as it is safe to do so. Then the control returns to the calling sequence.

Note that the SCR MOD sequence is also responsible for determining the state of the OS full flip-flop, OSF. As described in section 2.3.1.4, this flip-flop is used to determine whether the hardware registers are full or empty whenever the new SCR position equals the OSTR position. In order to perform this operation, the SCR MOD sequence resets OSF immediately before incrementing the SCR. Then before the SCR is permanently changed, the OSF setting logic is activated to check for the necessary conditions for OSF to be on and to set OSF if necessary.

The OS WRITE sequence takes a data cell stored left-justified in the LR (and DR if the cell is a double-word) and writes it into the Operand Stack. It also increments the SCR by the cell size so that it will again point to the start of the first unfilled cell.

The OS READ sequence first uses the SCR MOD sequence to position the SCR so it points to the beginning of the desired cell. This means that the SCR must be decremented by one cell size if the top operand is being read (CS2 = 1). If the read is a "store" type, i.e., the final SCR position is the same as the initial, or alternatively if the WAIT signal is on, the SCR MOD sequence is not allowed to change PR#13 even if there is a wraparound.

After the SCR value has been decremented, the new value is used to read out the desired cell, (i.e., the modified SCR value is fed to the byte selection circuit). If the data is to be returned in negative

form (i.e.  $NEG = 1$ ), the "gate false out" control is used for the OS. Otherwise the "gate true out" is used (see Section 2.3.4). If the NWTOS signal is on, the output is placed on the DB only. Otherwise it is also loaded left-justified into the LR (and DR if the cell is a double word). Note that a double word number will never be requested in 1's complement form.

Finally, if the access to the OS is of a "store" type, the SCR is incremented back to its original position. Note that in this case manipulation of PR#13 is again inhibited.

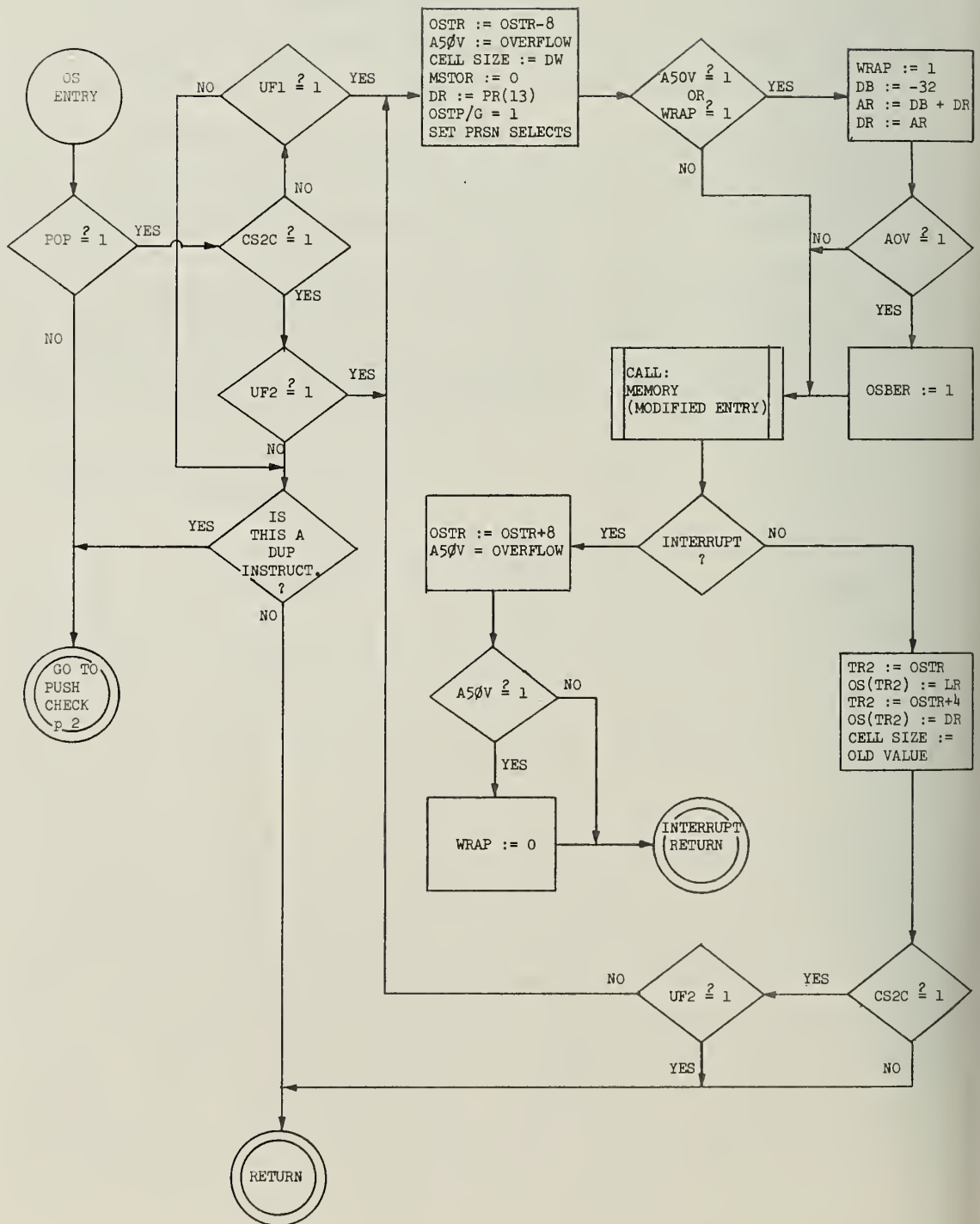
One persistent problem which occurs in using the OS is the problem of detecting the "top" and/or "bottom" of the storage area allocated to the OS. There are no built-in safeguards within the OS. There are no built-in safeguards within the OS logic itself since any procedure which was truly effective in the general case would involve a prohibitive amount of logic.

In order to provide a minimum amount of hardware protection several conventions have been specified. First of all, Operand Stack storage areas should always be allocated their own separate segments. This will keep the TP from accidentally spilling out of the stack area into an area where it does not belong. If an access beyond the confines of the OS storage segment is attempted, a bounds overflow interrupt will result.

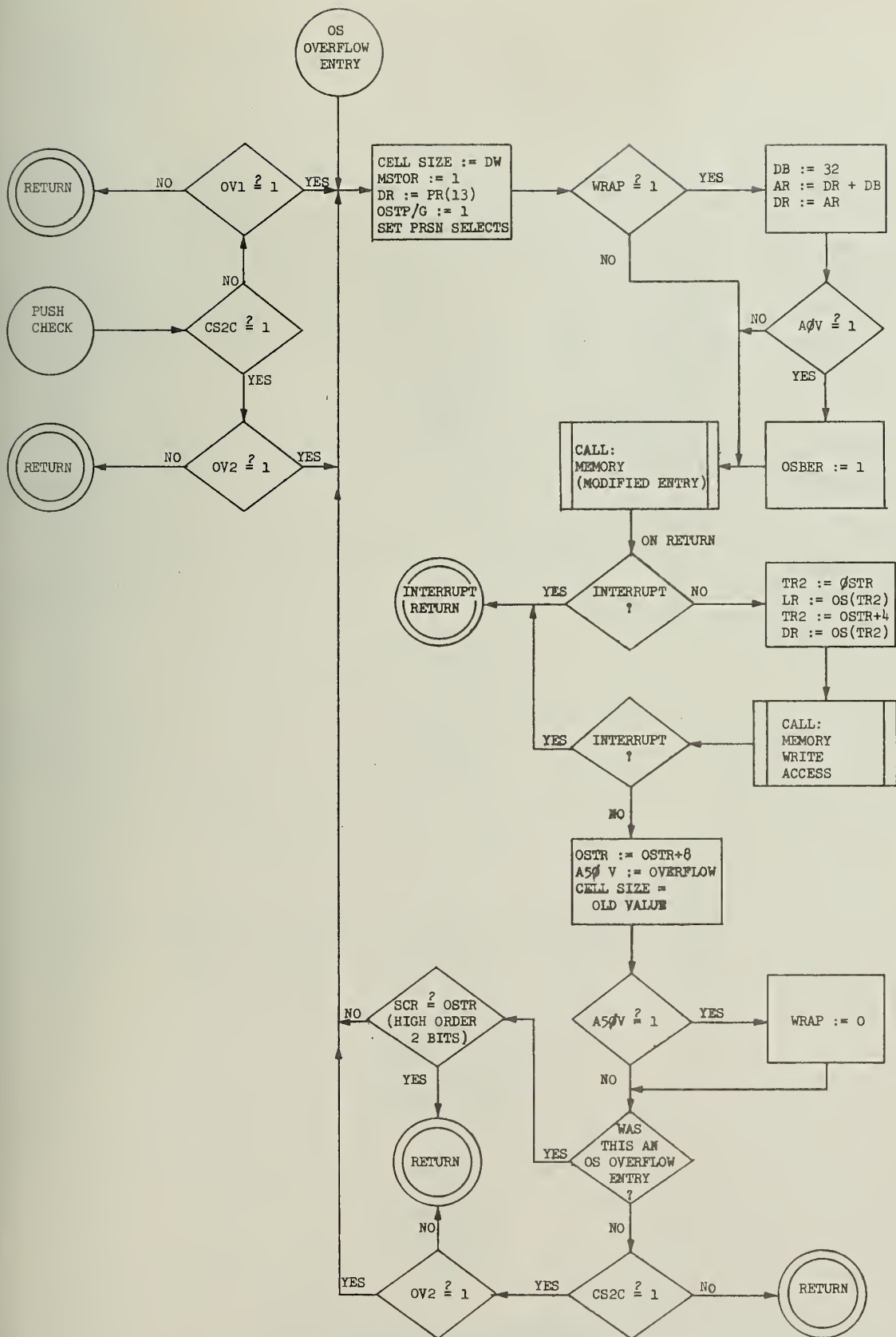
Even with this type of protection, there is another way that one might get into trouble. This problem is a result of the wraparound effect when pointer registers are incremented above  $2^{16}$  or decremented below 0. If an operand stack segment has been allocated 256 pages of core, the wraparound at either end of the OS segment will not cause a bounds overflow interrupt since both extreme addresses are within the segment. This wraparound would, in general, be a very undesirable effect and yet might very easily happen if due to some error the program loses track of how much data it has in the OS.

The solution to this problem involves checking the 32-bit adder overflow signal everytime PR#13 is incremented or decremented due to an SCR wraparound at the 0-32 boundary. If an overflow in the 32-bit adder occurs during this calculation, a PR#13 wraparound has occurred and the OS segment boundary has probably been fouled up.

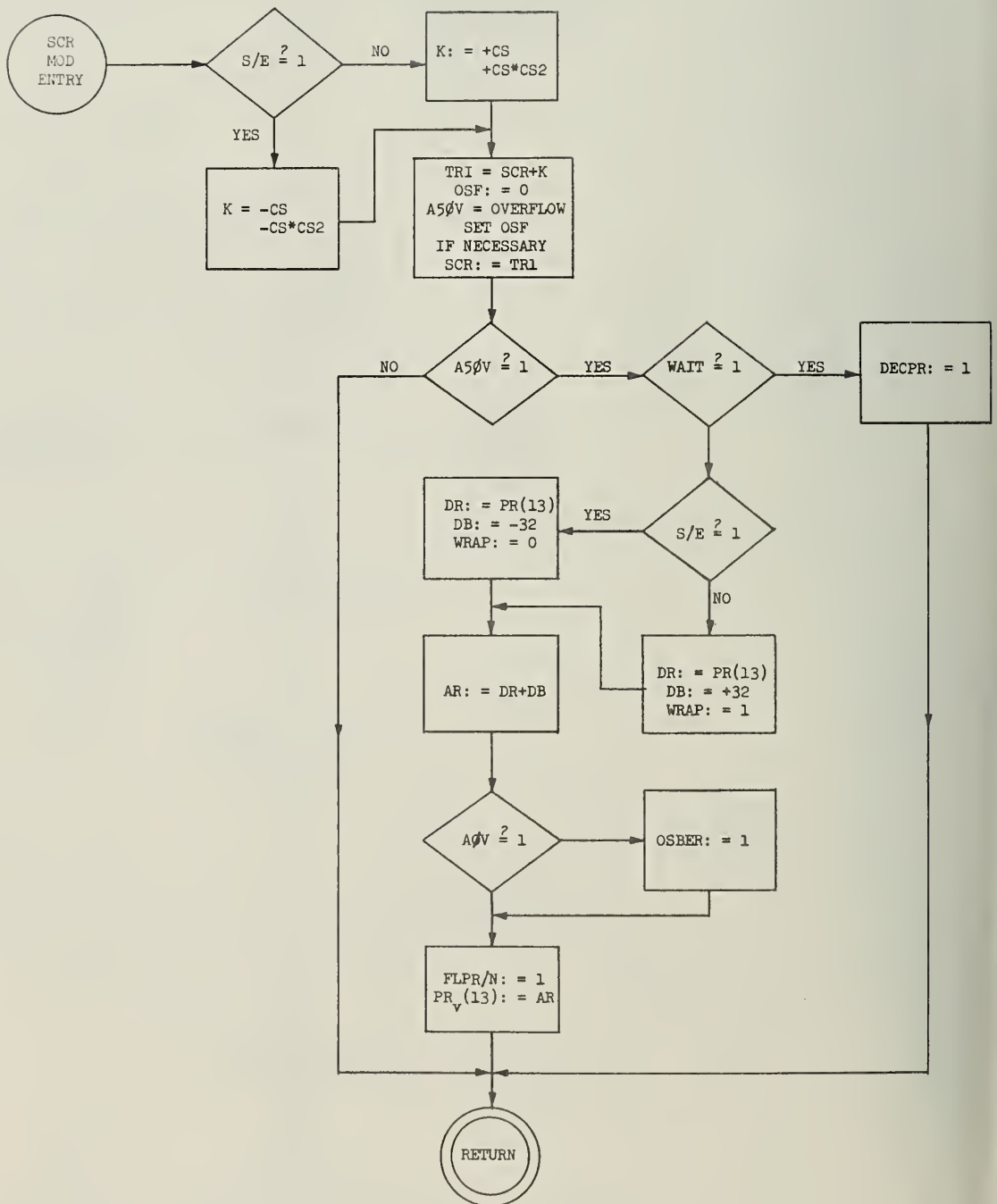
The above check is made in the SCR MOD sequence if the value of PR#13 is changed during its operation or whenever the calculation is performed if it does not occur during SCR MOD. It must also be made when OS entry uses  $PR\#13 \pm 32$  to access core. These checks will catch any wraparound attempt by the TP hardware during the execution of instructions involving the OS. However, it probably will not catch a wraparound caused by independent manipulation of PR#13 by the programmer.



Operand Stack - Basic Operations  
OS ENTRY Sequence Flow Chart

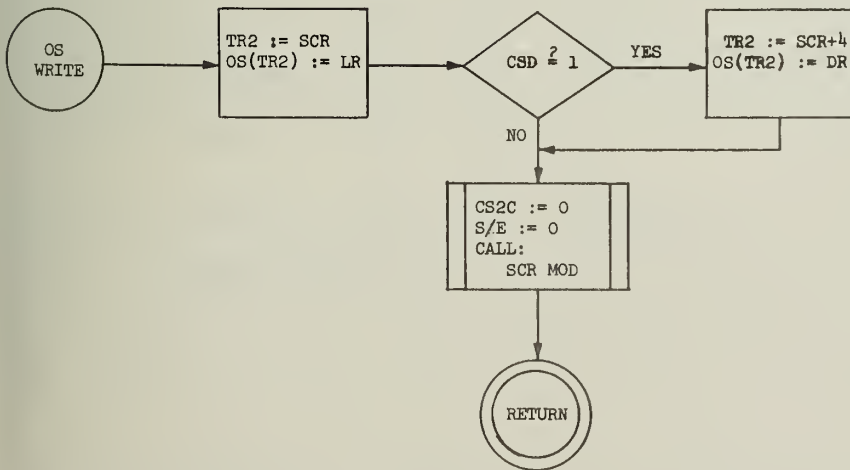
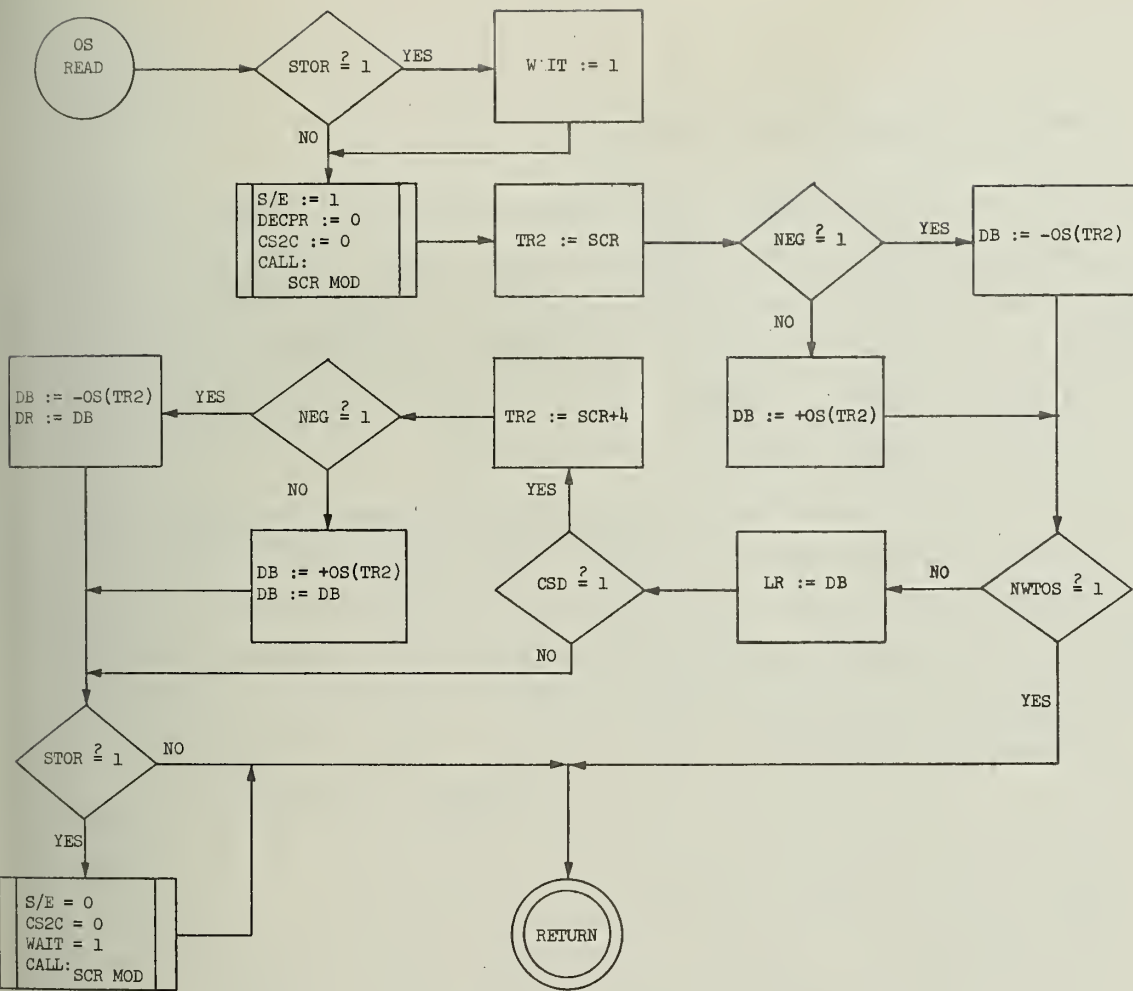


Operand Stack - Basic Operations  
OS ENTRY Sequence Flow Chart



Operand Stack - Basic Operations  
SCR MOD Sequence Flow Chart







#### 4.5.1.2 OS ENTRY Control Logic

The OS ENTRY control logic is governed through the use of one control flip-flop, POP, and one control signal, CS2C, both of which are set before entering the sequence. POP equals 1 if the OS sequence is one in which data must be read from the stack and 0 if data must be written into the stack. CS2C is 1 if 2 cells rather than one will be written into or read out of the stack.

The first part of the sequence consists of checking the control signals together with the overflow-underflow indicators to see if anything must be done (refer to Figure 4.5.1.2). The gate signals OSG1 through OSG6 are indicated on the flowchart at the end of the section. The decision logic involved is implemented in standard integrated circuits.

The decision logic eventually generates either a return signal, in which case the sequence is done, or it activates OST1 which sets the OVLOOP/UNLOOP flip-flop to the type of operation to be performed (i.e. overflow correction, in which case data must be taken from the memory and stored in the hardware registers or underflow correction, in which case data must be taken from the hardware register and stored in memory). If either type of correction is necessary, control will then be transferred to OST2. The OVLOOP and UNLOOP signals are used to control the flow of the control sequence and the operations which are performed.

Since it is impossible to have both an underflow and overflow condition arise on the same access to the OS ENTRY sequence, and since the operations involved in these two cases are quite similar, it is advantageous to use the same control points for both operations whenever possible.

In control points, OST2 and OST15, the Operand Stack full flip-flop OSF is reset. There is never any need to set the OSF in the OS ENTRY sequence since during corrections for possible stack overflow or underflow the OSTR is always moved away from the SCR.

$(\text{OSENSTRT} \cdot \text{POP}) \cdot \text{CS2C} \cdot \text{UV2} \vee$   
 $(\text{OSENSTRT} \cdot \text{POP}) \cdot \overline{\text{CS2C}} \cdot \text{UV1} \vee \text{OSG1}$

underflow  
segment

$(\text{OSENSTRT} \cdot \overline{\text{POP}}) \cdot \text{OV2} \cdot \text{CS2C} \vee$   
 $(\text{OSENSTRT} \cdot \overline{\text{POP}}) \cdot \text{OV1} \cdot \overline{\text{CS2C}} \vee$   
 $(\text{OSENSTRT} \cdot \text{POP}) \cdot (\overline{\text{UV2}} \cdot \text{CS2C}) \cdot \text{DUP} \cdot \text{OV2} \vee$   
 $(\text{OSENSTRT} \cdot \text{POP}) \cdot (\overline{\text{UV1}} \cdot \overline{\text{CS2C}}) \cdot \text{DUP} \cdot \text{OV1} \vee$   
 $(\text{OSG4} \vee \text{OSG6}) \vee \text{OVENSTRT}$

overflow  
segment

$(\text{OSENSTRT} \cdot \text{POP}) \cdot (\overline{\text{UV2}} \cdot \text{CS2C}) \cdot \overline{\text{DUP}} \vee$   
 $(\text{OSENSTRT} \cdot \text{POP}) \cdot (\overline{\text{UV1}} \cdot \overline{\text{CS2C}}) \cdot \overline{\text{DUP}} \vee$   
 $(\text{OSENSTRT} \cdot \text{POP}) \cdot (\overline{\text{UV2}} \cdot \text{CS2C}) \cdot \text{DUP} \cdot \overline{\text{OV2}} \vee$   
 $(\text{OSENSTRT} \cdot \text{POP}) \cdot (\overline{\text{UV1}} \cdot \overline{\text{CS2C}}) \cdot \text{DUP} \cdot \overline{\text{OV1}} \vee$   
 $(\text{OSENSTRT} \cdot \overline{\text{POP}}) \cdot \overline{\text{OV2}} \cdot \text{CS2C} \vee$   
 $(\text{OSENSTRT} \cdot \overline{\text{POP}}) \cdot \overline{\text{OV1}} \cdot \overline{\text{CS2C}} \vee$   
 $(\text{OSG2} \vee \text{OSG3} \vee \text{OSG5})$

return

Figure 4.5.1.2 - OS ENTRY  
Input Decision Logic

Note that in OST<sup>4</sup> the AOV signal is checked and, if active, is used to set the Operand Stack Bounds Error flip-flop, OSBER. This is one of the checks for PR#13 wraparound which was discussed at the end of Section 4.2.2.7.1.

Another point to notice is that in OS underflow before control step OST<sup>9</sup>, it is not necessary to load the OSTR into TR<sup>2</sup>. This is because in the underflow case, TR<sup>2</sup> was previously loaded with the same contents as the OSTR in control step OST<sup>3</sup> and no operations have been performed on either in the meantime. Thus they have the identical contents.

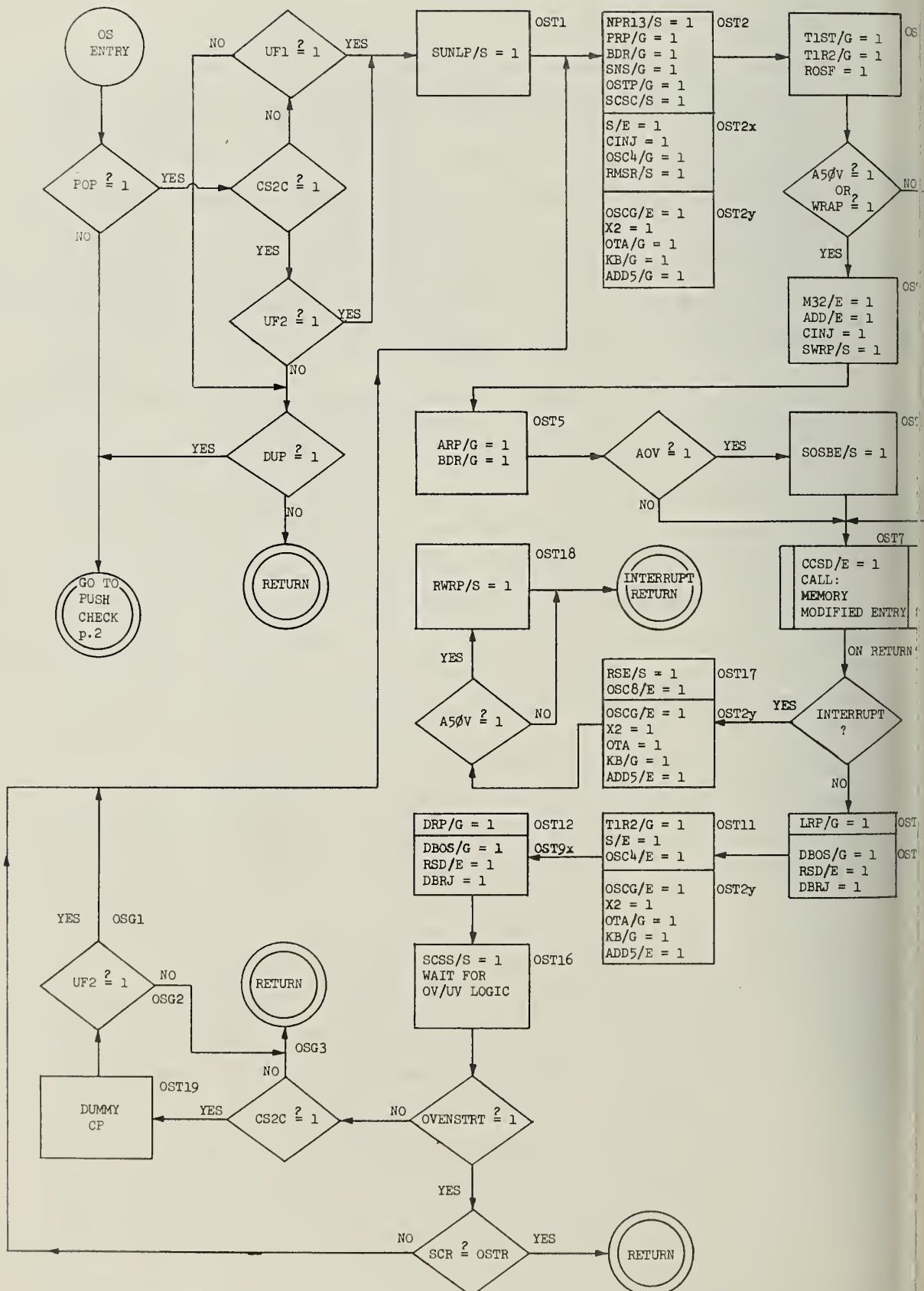
In control point OST<sup>16</sup> the cell size selector is set to the CSS/E signal so that the Overflow-Underflow logic in the OS Control group can determine whether another cycle is necessary. It should be noted that at the beginning of the sequence the CS selector has already been set by the calling sequence. Then the selector is changed during the memory access so that the control logic can force a cell size of double word to be used. If only one cycle is performed, the sequence will perform an exit with the selector enabling CSS/E and it will be up to the calling sequence to reset the cell size selector if necessary.

If a second cycle is necessary, it will be assumed that the cell size is determined by CSS/E. Since all the cases which do not have cell sizes determined by CSS/E also do not need two double word cells (which is the only case which might need a second cycle) this is a safe assumption.

Control point OST<sup>19</sup> is a dummy control point. It is used in order to provide proper switching in the decision logic which uses the OS overflow-underflow signals. Since these signals will be changing during the operation of control point OST<sup>16</sup> (due to the change of state in the cell size signals caused by turning on SCSS/S) it is not possible to have decision logic immediately following OST<sup>16</sup>, which depends on these changing signals (see section 4.A.2). Thus OST<sup>19</sup> is used to provide an advance out signal which can be used to activate the OS overflow-underflow checking logic.

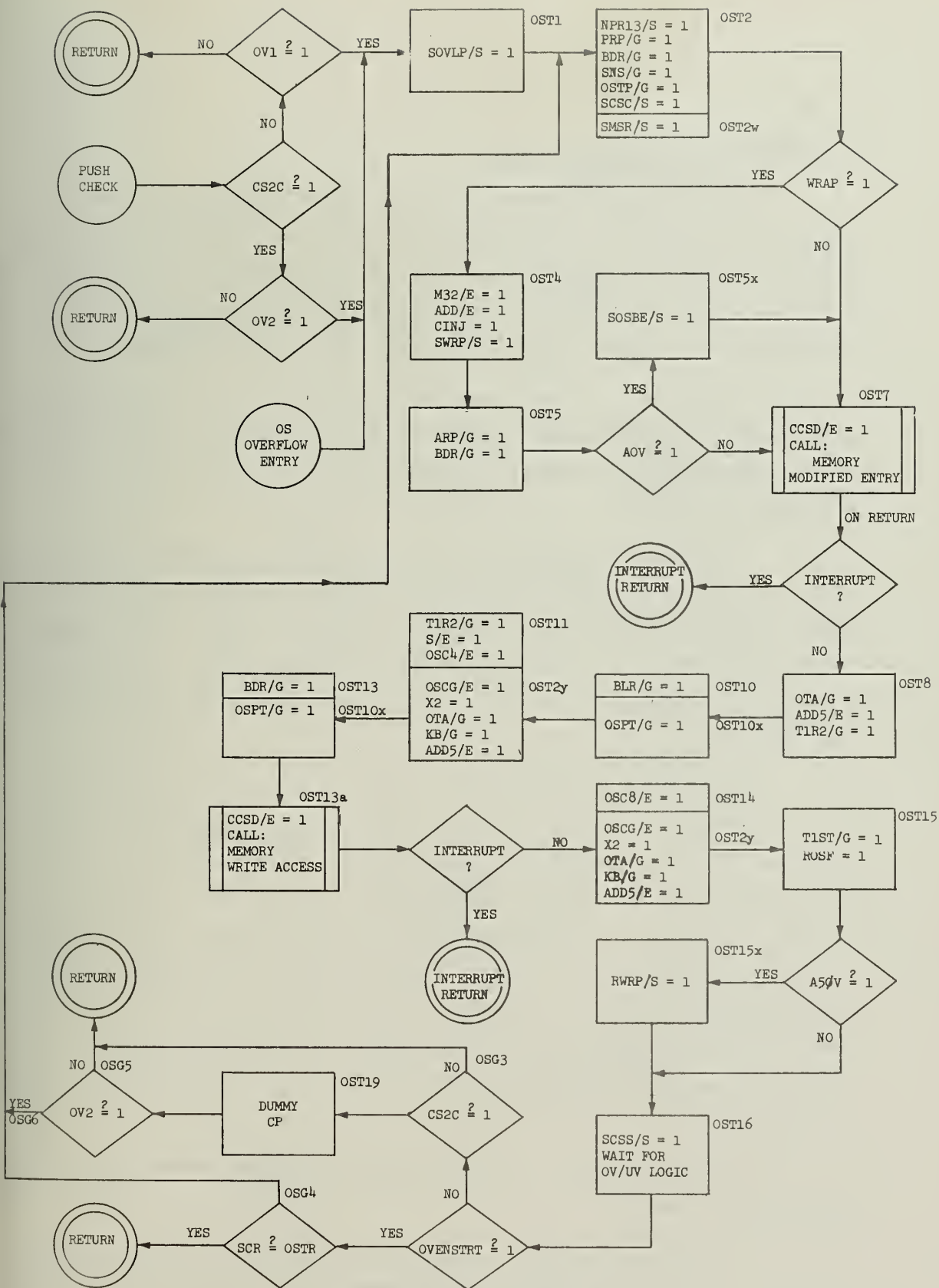
In the original design of this control logic the number of control points was diminished by combining OST1 and OST14, OST2 and OST15, OST8 and OST11, and OST9, OST10 and OST13. However, the added complexity of the resulting logic around these control points almost negated the savings in control point logic. In addition, the speed of the circuit was reduced and the circuit as a whole was much more difficult to understand. For these reasons, the present version with additional control points was selected.

The other aspects of this control logic are fairly straightforward.



OS ENTRY - Control Point Flow Chart





OS ENTRY - Control Point Flow Chart

#### 4.5.1.3 SCR MOD Control Logic

The SCR MOD sequence, whose flowchart is shown at the end of this section, is used to change the value of the SCR and to provide corrections to PR#13 in the case of SCR underflow or overflow. There are three control signals, S/E, X2 and WAIT which are used for control of the sequence. Signals X2 and CS are global to the sequence.

When S/E is 1, the negative constants in the 5 bit Adder Constant Generator are enabled. Thus S/E must be held at 1 for the duration of the sequence if decrementing of the SCR is desired and at 0 if incrementing is desired.

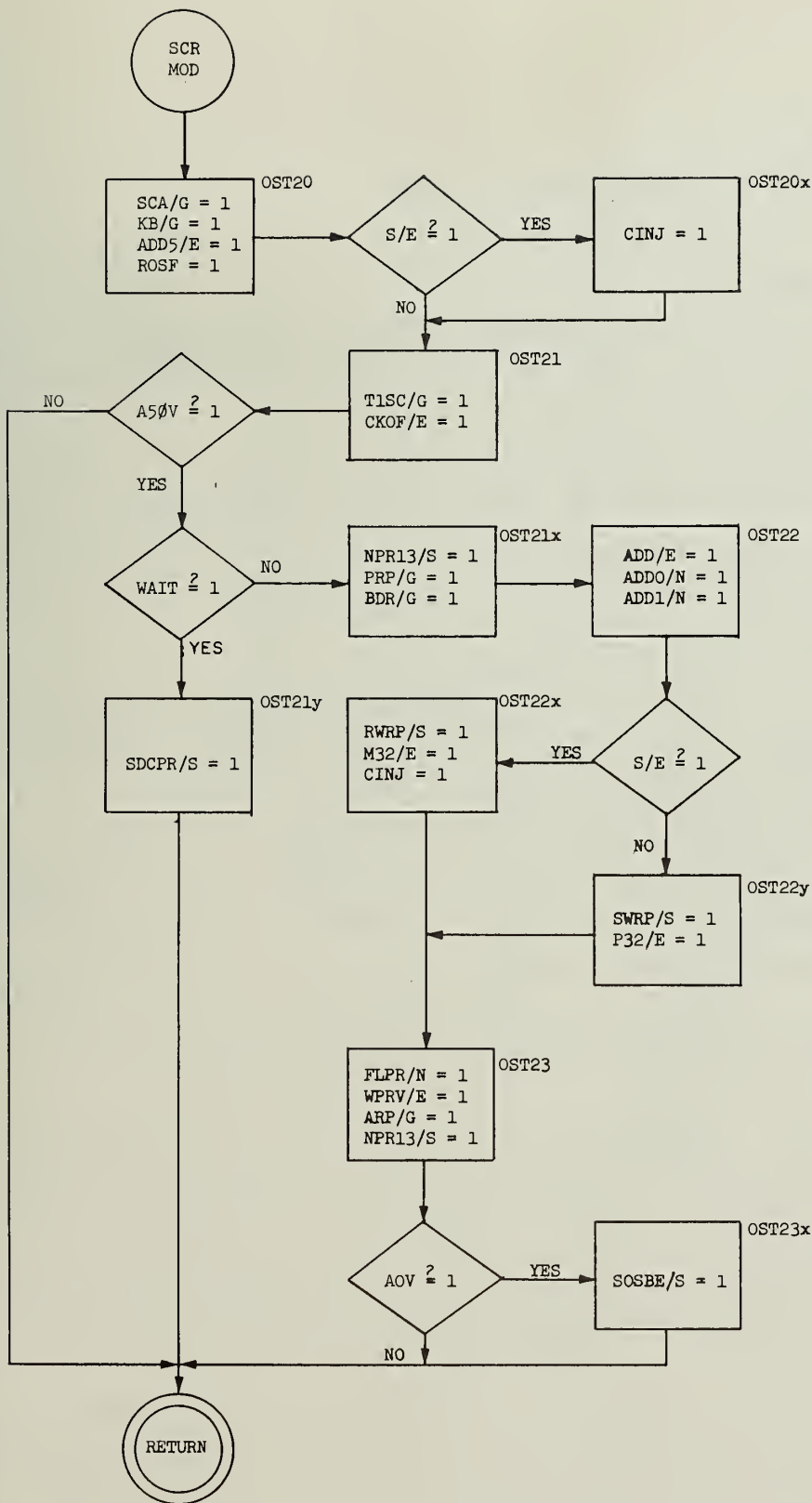
X2 must be turned on if it is desired to modify the SCR by two cell sizes instead of one. This is really just another signal to the 5 bit Adder Constant Generator. In actual use however, it is often turned on by a calling control point for an OS READ Operation. This causes the OS READ sequence to read out the next-to-the-top cell in the OS instead of the top cell.

WAIT is a special control signal which is set to 1 in cases where it is undesirable to adjust PR#13 in case of SCR underflow or overflow. This situation will arise whenever iterim data is being stored in either the DR or AR, which must be used to modify PR#13, or in a case in which the SCR is only being moved temporarily and will be moved back to its original position almost immediately. Thus if the WAIT signal is on and an underflow or overflow condition occurs, a special flip-flop, DECPR, will be set and PR#13 will be untouched. This will enable the calling sequence to know that PR#13 should be adjusted, if desired, as soon as the DR and AR can be used again.

The WAIT signal is often turned on before an access to OS READ or OS WRITE is made.

Note that in OST23 a check for wraparound in PR#13 is made. If there is an adder overflow, the Operand Stack Bounds Error flip-flop is set (see Section 4.5.1.1 for further details).





SCR MOD Sequence Control Point Flow Chart

#### 4.5.1.4 OS READ Control Logic

The OS READ sequence is used to read data out of the OS hardware registers into the LR (and DR in the case of a double word cell). There are two control flip-flops, NWTOS and STOR, and one control signal, NEG, which must be set prior to starting the sequence. In addition, the WAIT control signal for the SCR MOD sequence may also be turned on if it is necessary to postpone any possible manipulation of PR#13 (e.g. it may be necessary to use the DR or AR for temporary storage during the operation of OS READ; if the WAIT inhibit were not on and SCR MOD created a "wraparound" on the SCR, then PR#13 would be incremented by 32 and the DR and AR contents would be destroyed in the process).

STOR is used to indicate that the SCR will not be permanently changed by the OS READ sequence. Therefore if wraparound occurs during the SCR MOD sequence no change in PR#13 is necessary since it would only have to be changed back at the end of the OS READ sequence. STOR is used to activate the WAIT signal during SCR MOD if the WAIT signal has not been turned on already. It also is used to decide if the SCR should be incremented back to its original position at the end of the sequence.

NEG is set to one if the data is to be read out of the OS in complement form. Otherwise it is set to zero.

NWTOS (no write from the OS) is set to one for the special case when the data from the OS is only to be placed on the distribution bus and not loaded into the LR. This case will only occur for non-double word cells and for STOR = 0. This causes a complication in task OST32 since if NWTOS is one, the task must remain on until after the OS READ sequence has returned to the calling logic.

In order to realize OST32 the design shown in Figure 4.5.1.4 is used. Note that it is a more or less standard control point except that the output of the delay circuitry is not fed directly back to the

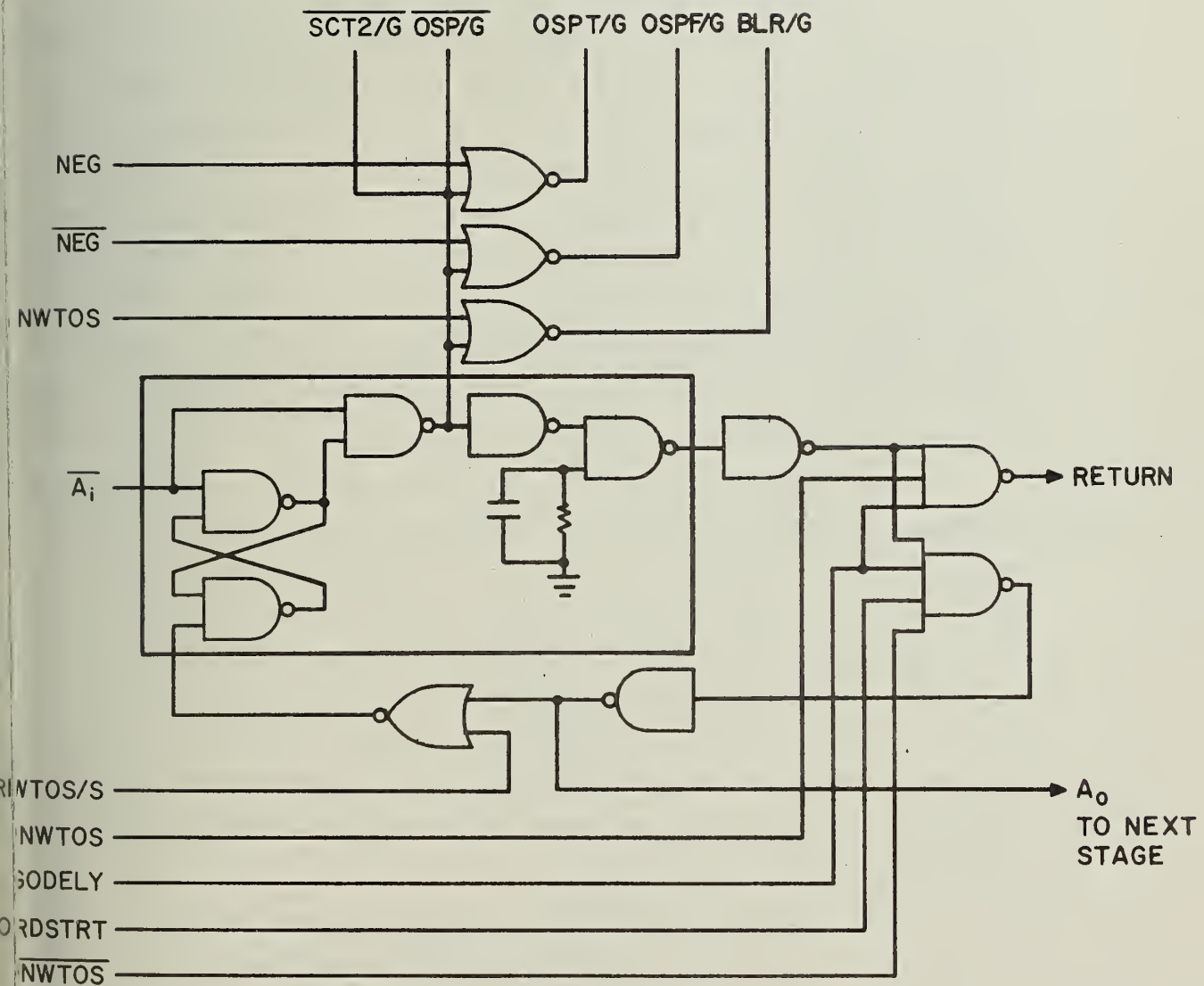


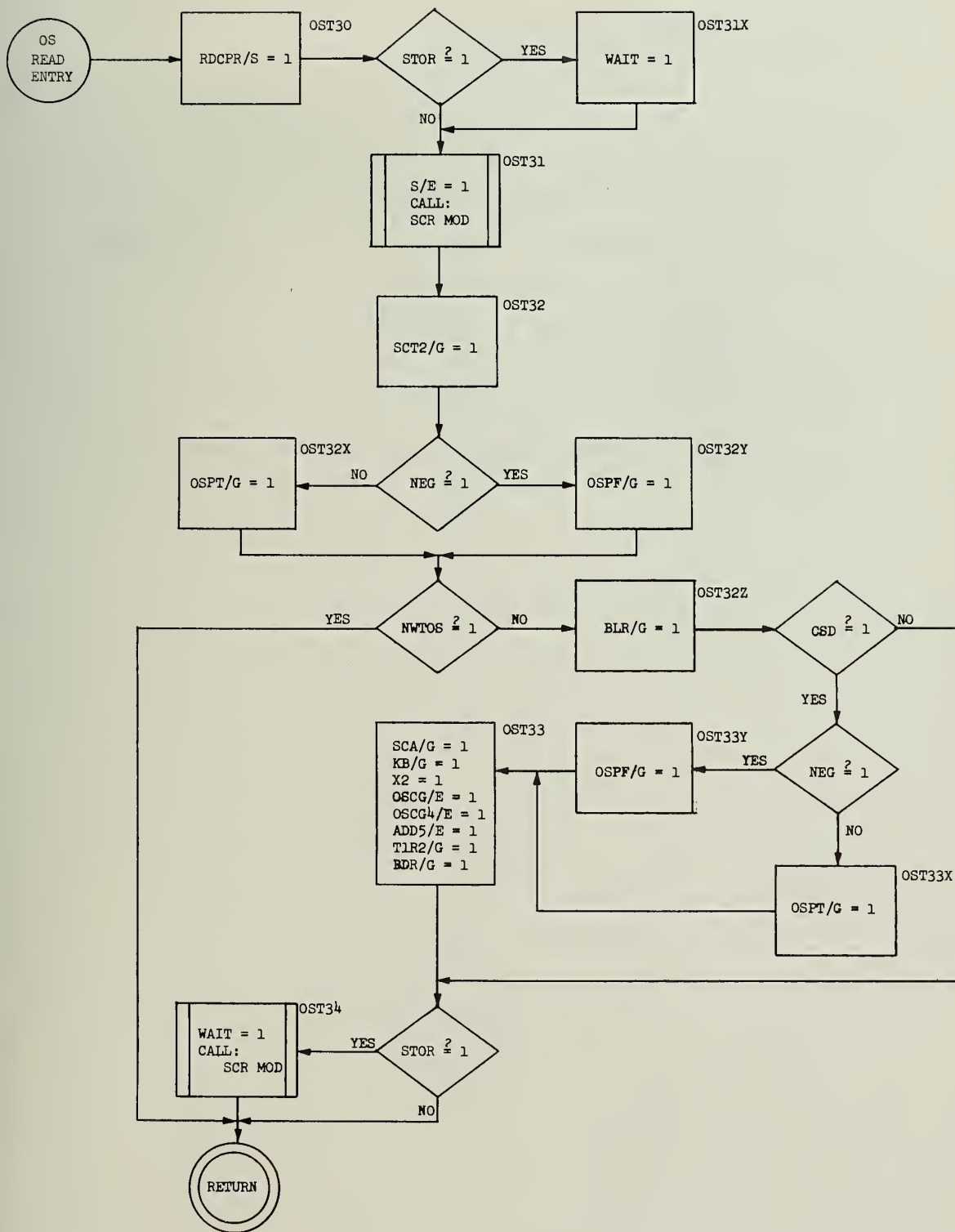
Figure 4.5.1.4 - Control Point OST32

control point flip-flop. Instead it is fed to some decision logic which causes a return without resetting the control point if NWTOS = 1. If NWTOS = 0 and OSRDSTRT = 1 then the control point is reset and the sequence continues.

The need for OSRDSTRT may not be obvious. The reason it is used is that if NWTOS = 1 (and thus the OS READ sequence returns without resetting the control point), the OST32 task signal will remain on so that the calling sequence can use the data on the DB. Eventually the calling sequence will have to reset OST32 in order to use the permuter for other things. This is done by turning on RNWTOS/S which resets NWTOS and also resets OST32. When NWTOS goes to "0" the advance signal to the next stage after OST32 would be turned on (a highly undesirable effect) since the output of the delay circuit is still on. However by AND'ing OSRDSTRT to these signals we insure that the advance out signal will only be propagated in the case where the OS READ sequence is active.

A further word might be said about RNWTOS/S. Before calling OS READ the calling sequence will ordinarily have to reset the NWTOS flip-flop if it desires NWTOS to be '0'. Oftentimes, the setting or resetting of a control flip-flop such as NWTOS is performed using the same calling control point flip-flop which actually initiates the sequence. In the present case, however, this must not be done since to do so would cause OST32 to remain in a reset condition for the whole sequence. This would obviously cause a severe error in the OS READ operation. If NWTOS must be reset before calling OS READ, it must be done prior to the actual call.

In control point OST27 it should be remembered (see Section 2.3.1.2 on the OS Constant Generator) that the X2 signal is turned on in order to allow the generation of a constant directly from the control and not to multiply that constant by two. Thus when OSC4/E is used with OSCG/E and X2, the constant appearing on the K bus is +4, not +8.



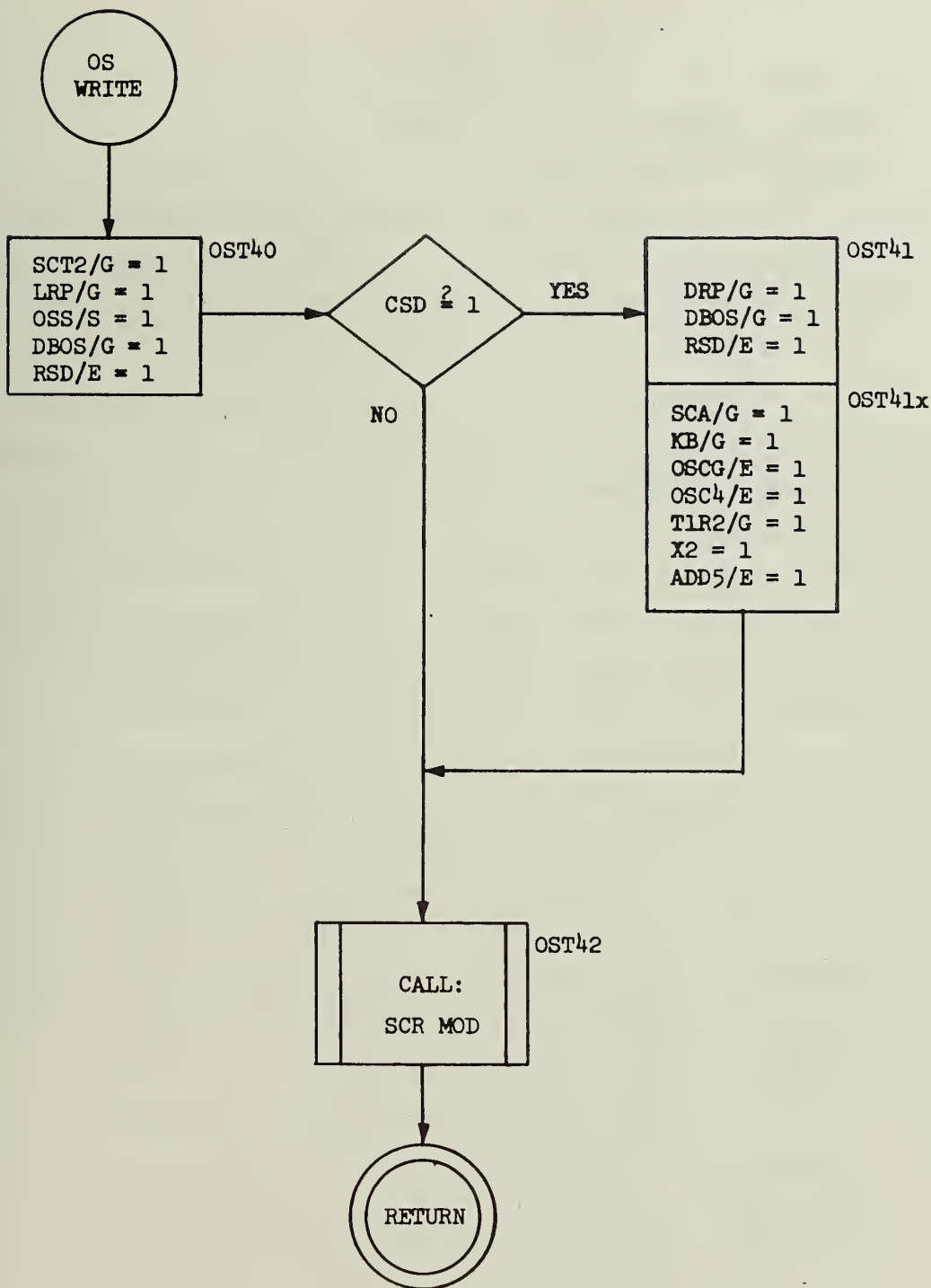
Operand Stack - OS READ Sequence Control Point Flow Chart

#### 4.5.1.5 OS WRITE Control Logic

The OS WRITE sequence is used to fill the OS hardware registers with data which has previously been loaded into the LR (and DR in the case of a double word cell). There are no control signals or flip-flops for the OS WRITE sequence itself. However the WAIT control signal for the SCR MOD sequence should be turned on if the OS WRITE sequence is returning the SCR to a previous position from which it was moved while the WAIT signal was on. Specifically, if the SCR is decremented with the WAIT signal on (because it will shortly be incremented back), then the WAIT signal must also be on when the SCR back incrementation is done.

The OS WRITE sequence itself is very straightforward.







## 4.5.2 Supplementary OS Sequences

### 4.5.2.1 Supplementary OS Sequence Descriptions

The Supplementary OS sequences consist of OS CLEAR and OS INITIAL which are used, respectively, to clear out the OS hardware registers by storing them in core memory and to partially load the hardware registers when the top of the OS is moved by changing PR#13.

In order to understand the need for these sequences several possibly dangerous situations should be considered. For example, as was previously discussed (Section 2.3), the Operand Stack operates in the area defined by PR#13. However, at any given time an indeterminate number of bytes of the OS are actually stored in the fast registers of the TP. Thus if a programmer were to use PR#13 to access a cell in the stack directly (i.e. not through an Operand Stack primitive instruction), he might get invalid data if the cell were actually in the TP and not in core.

Another problem arises in the situation in which the programmer wishes to change the location of his Operand Stack (i.e. modify PR#13). In this case the TP fast registers must be cleared of any valid Operand Stack data they may contain before the stack is "moved" and afterwards the SCR and ØSTR must be initialized to the new ØS position. A change in location of the OS may be effected either by changing the PR's value or its associated segment name, but not by merely storing the value of PR#13

Finally, if during an imprimitive instruction, the name of PR#13 is permuted, we have the same type of situation as that in which the contents of PR#13 are modified, i.e. the location of the OS will change. Then in this case, too, the TP registers must be cleared and initialized.

The various occurrences of these situations can be broken down into four types:

- 1) Changing PR#13 by a PR Modification: in this case a check is made at the beginning of the sequence to see if PR#13 is the one being modified. If it is, the OS is cleared and the OSINT flip-flop is set. This signifies that before the OS is used again it must be initialized. The initialization is usually done at the end of the instruction if not before.
- 2) Name Permutation of PR#13: if the name of PR#13 is to be changed by an imprimitive instruction, the OS is cleared and OSINT is set. At the end of the instruction, if OSINT is on, either because of a name permutation or a PR#13 modification, the OS is initialized.
- 3) Using PR#13 in an Address Construction: if PR#13 is used to construct an address (other than for adjusting the stack because of overflow or underflow), an OS clear must be performed before constructing the address. The Modified Memory Access entry skips this check; this entry is mainly used by the control, and the control does not make accesses straight into the middle of the stack. The OSINT flip-flop is unchanged.
- 4) Instructions with modify value or segment name of PR#13: in any of these instructions the stack must be cleared before the old value is destroyed and then initialized once the new value has been entered.

The OS CLEAR sequence flow chart is given at the end of this section. The clearing operation is effected by entering the Operand Stack Overflow subsequence and executing it until the stack is empty. Note that if the top of the stack does not end on a double word boundary one or more bytes just beyond the top of the stack in core will be overwritten with garbage. This may or may not matter depending on the particular problem.

There is also one other problem which might occur during a clearing operation. Suppose that a new Operand Stack is to be started in the middle of a double word at address  $\alpha$  (see Figure 4.5.2.1/1 below). In this case the OS will be initialized with operands A and B in the hardware registers, since initialization always starts at a double word boundary. If the hardware registers subsequently become full, the A and B operands may have to be put back into core to make room.

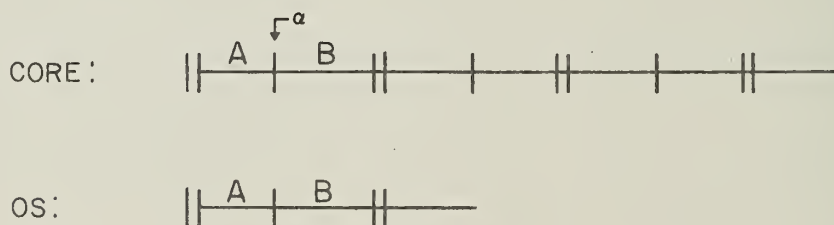


Figure 4.5.2.1/1

But what if in the meantime 'B' was overwritten in core by using a PR other than PR#13? The new data in core will also be overwritten and 'B' will be the final result. This is perfectly legitimate and demonstrates one of the dangers of working on data near the top of the OS without using PR#13 to do so.

However, since  $\alpha$  is the "bottom" of the OS, it would be perfectly reasonable for some operation to modify A without using PR#13 (which would cause the stack to be cleared). In this case if the hardware registers were emptied into core the old value of A would be restored and a perfectly legal new value would have been destroyed.

There appears to be two solutions to this problem: 1) all newly formed Operand Stacks should begin on double word boundaries, or 2) there should always be a 1 double word buffer between the beginning of a new  $\emptyset$ S and previous block of data in core. Either of these would get rid of the problem for new stacks but the programmer must still be careful when he tries to manipulate data in the OS without using  $\emptyset$ S instructions or PR#13.

The  $\emptyset$ S INITIALIZE Sequence proceeds according to the following scheme (see flow chart at the end of this section):

- 1) When the new value for PR#13 is gated into the pointer register, its low order 5 bits (which are to be found on the low order bit positions of the DB) are simultaneously gated into the Stack Control Register (SCR). The high-order 2 of these 5 bits are also placed into the Operand Stack Top Register (OSTR). Thus the SCR defines the first empty cell in the Operand Stack to be the one whose address is the new PR#13 value. The stack full flip-flop (OSF) is set to 0.
- 2) If the low order 3 bits of the SCR are "000", the initialization is complete without further action. This means that the initial top of the Operand Stack was on a double word boundary and thus after stack initialization, the fast registers of the  $\emptyset$ S are empty.
- 3) If the low order 3 bits are not equal to "000", the double word in core starting at the double word boundary given by PR#13 (with the low-order 3 bits set to zero) is loaded into the Operand Stack fast registers starting at the register double word given by  $\emptyset$ STR. In this case, the initialized "hardware" stack contains between 1 and 7 valid Operand Stack bytes located between the positions of the  $\emptyset$ STR and the SCR. In actuality this may or may not be real  $\emptyset$ S data depending on whether the new PR#13 value represents the continuation of an old  $\emptyset$ S or the start of a new one. At any rate this concludes the initialization sequence.

Figure 4.5.2.1/2 shows two examples of stack initialization.

NEW PR #13 VALUE 

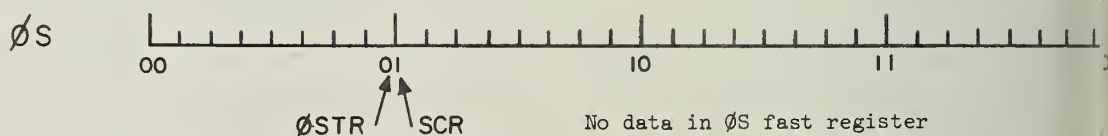
0	0	1	1	0	1	1	0	1	1	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

NEW SCR VALUE 

0	1	0	0	0
---	---	---	---	---

NEW ØSTR VALUE 

0	1
---	---



NEW PR #13 VALUE 

0	0	1	1	0	1	1	0	1	1	0	0	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

NEW SCR VALUE 

0	1	1	1	0
---	---	---	---	---

NEW ØSTR VALUE 

0	1
---	---

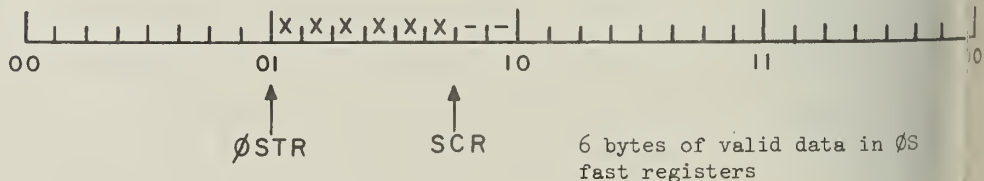
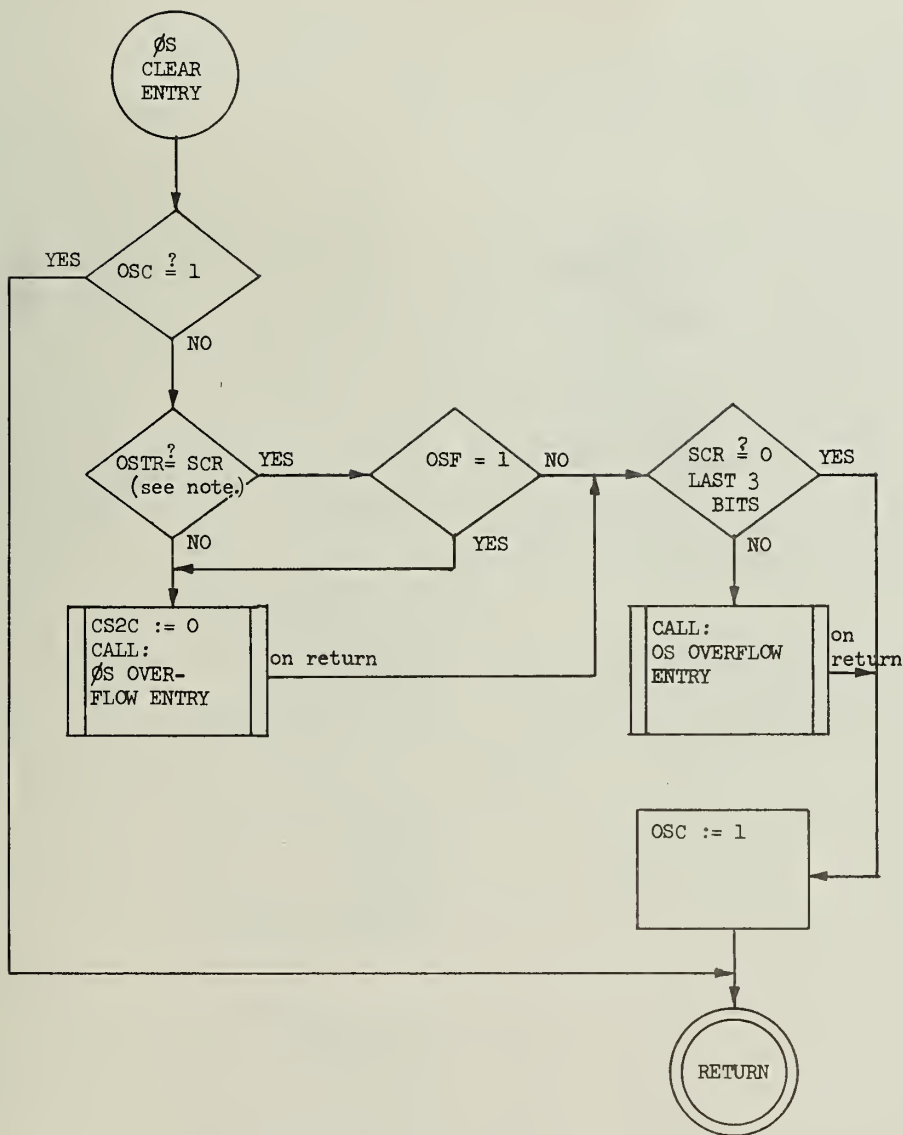


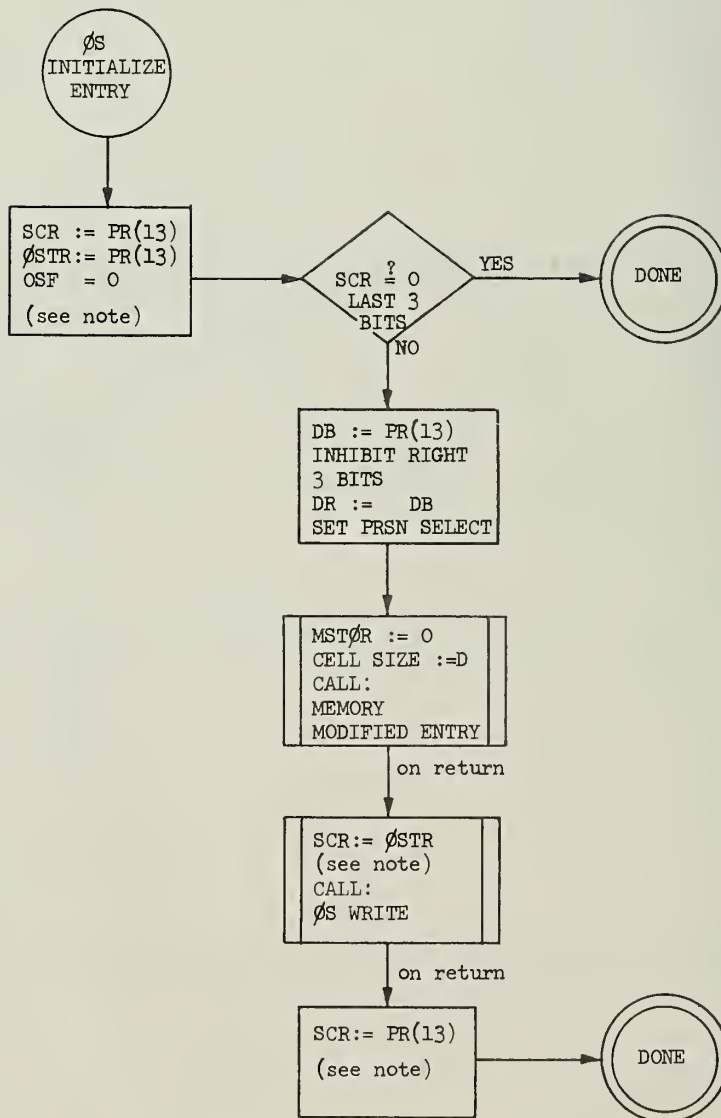
Figure 4.5.2.1/2 - Operand Stack Initialization Examples





NOTE: In data transfers in which the registers involved are not the same size, only those bits in the larger which have corresponding bit positions in the smaller are transferred. In equality only the common bit positions are compared.

Operand Stack Clear Sequence Flow Chart



NOTE: In data transfers in which the registers involved are not the same size, only those bits in the larger which have corresponding bit positions in the smaller are transferred. In equality only the common bit positions are compared.

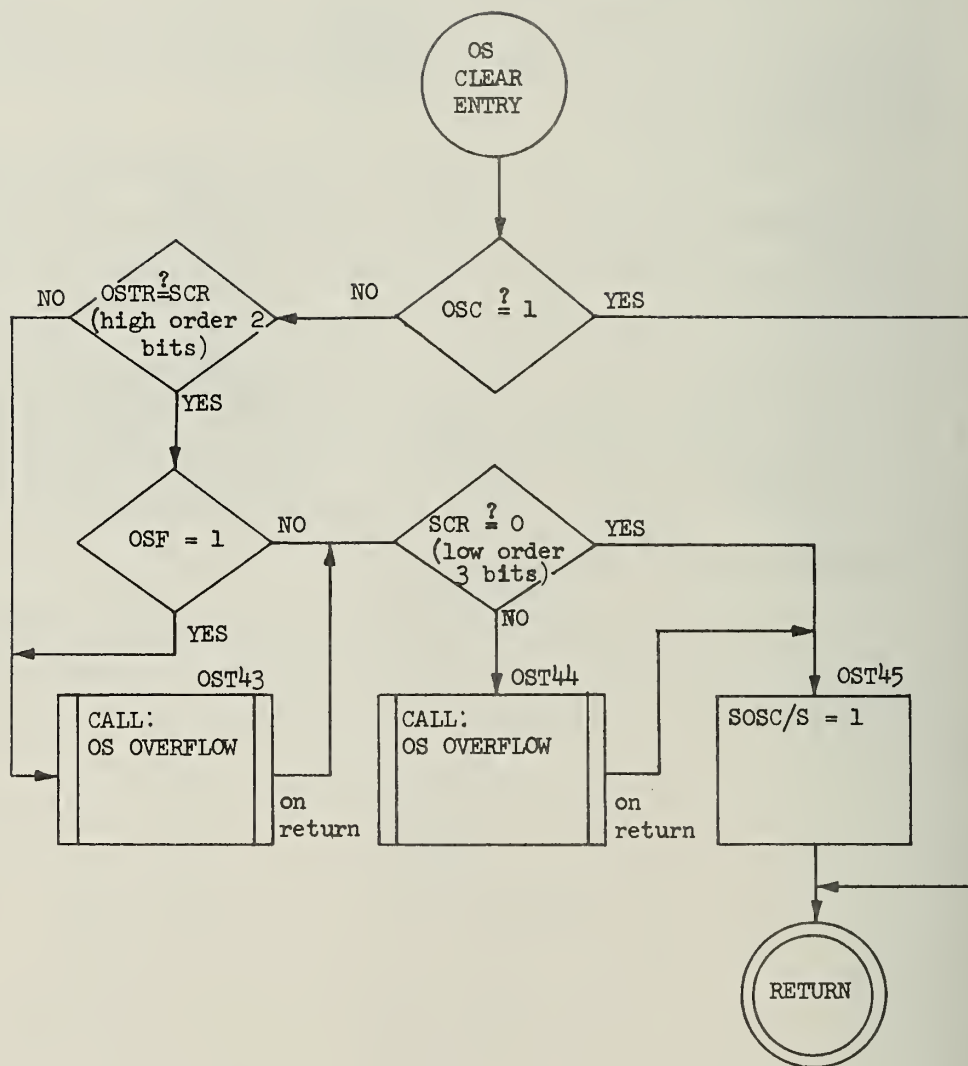


#### 4.5.2.2 OS CLEAR Control Logic

The OS CLEAR control logic is very short but slightly confusing at first since it is highly overlapped with the logic from the OS ENTRY sequence.

If OSC is 1, the Sequence returns immediately since the OS has already been cleared sometime previously. This check may be redundant in many cases because there have been so many changes in this logic that not all of the sequences have been updated to the latest changes. However if OSC = 1 and if the high order 2 bits of the SCR and the OSTR are not equal, or if equal and the OS is full, then the overflow section of the OS ENTRY sequence is executed, beginning at control point OST1. Note that this sequence will be continually executed and the OSTR is incremented until these bits are equal.

Next the low order 3 bits of SCR are checked for zero, and if they are not the overflow sequence is executed in final time. Finally, OSC is set to 1, Then the sequence returns.

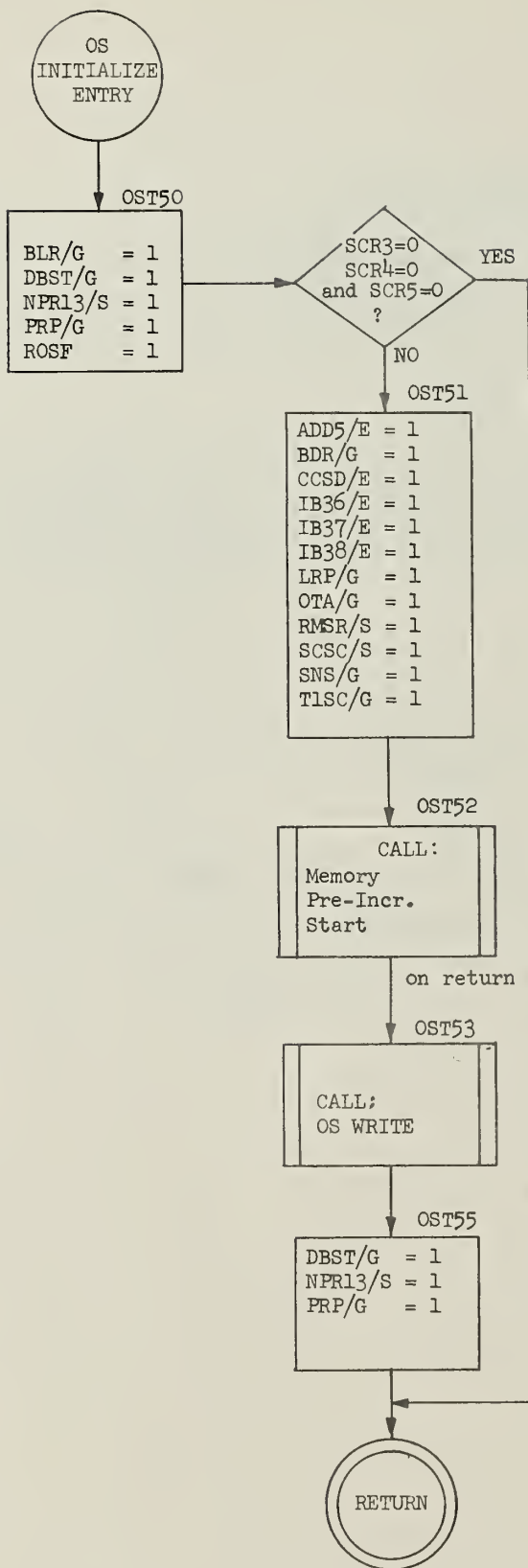


Operand Stack - Supplemental Operation Sequences  
OS CLEAR Control Step Flow Chart

#### 4.5.2.3 OS INITIALIZE Control Logic

The OS INITIALIZE sequence is used to load up the hardware Operand Stack in the TP and initialize the SCR and OSTR whenever a new value is loaded into PR#13. There are no control flip-flops to set prior to entering this sequence.

It should be noted that in the memory sequence call the cell size selector is set to the control signal selection. Therefore any sequence which uses OS INITIALIZE must reset the cell size selector after the return.



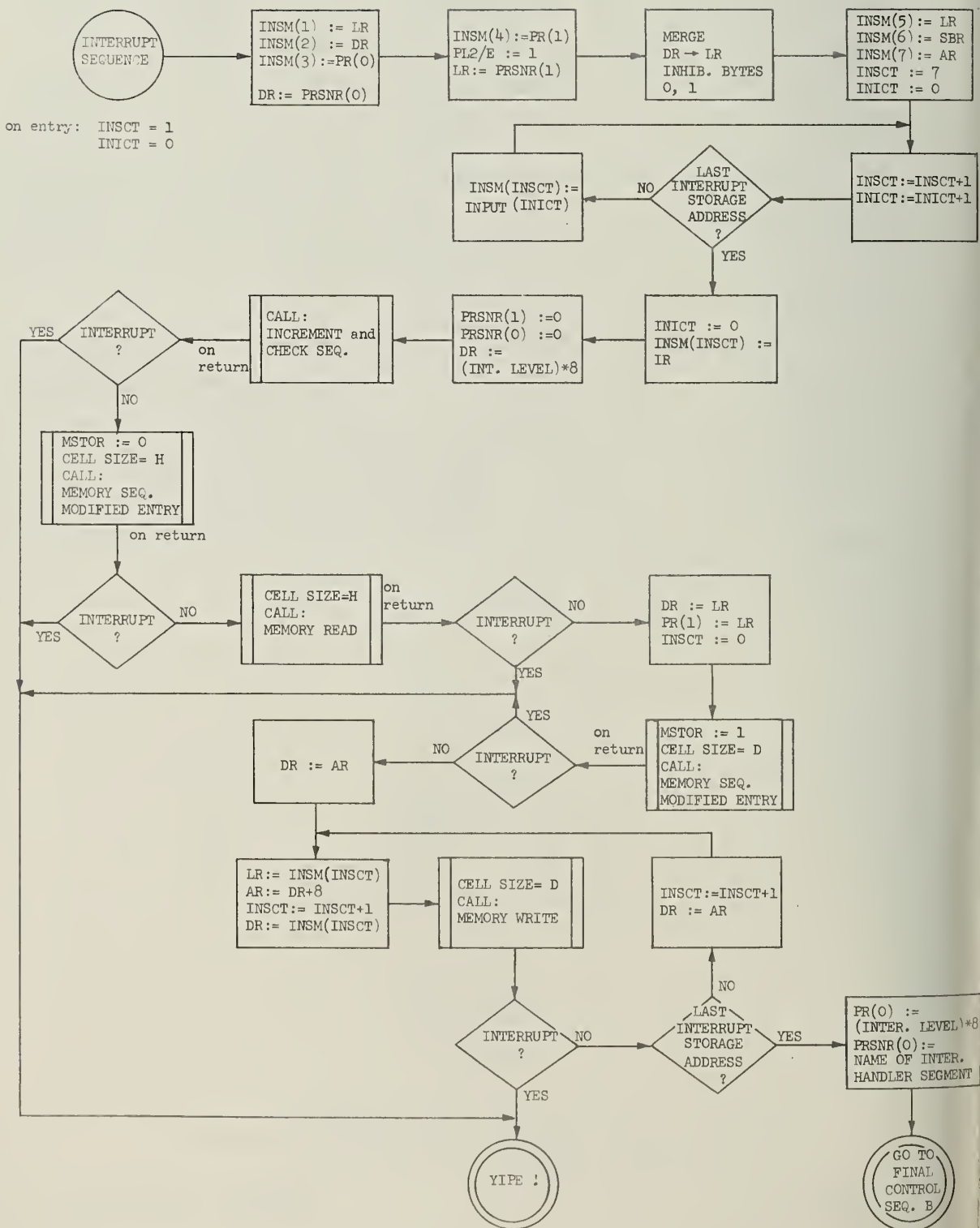
Supplementary OS Sequences OS INITIALIZE Control Step Flow Chart

#### 4.6 Interrupt Sequencing

The TP must be able to handle two types of interrupt conditions : local interrupts and distal interrupts. Local interrupts concern conditions originating within the TP or as a direct consequence of a TP command to the AU or PAU. These interrupts are also called traps. Examples include bounds overflow, loss of significance in a floating point AU operation, or an illegal plane address in a PAU instruction.

Distal interrupts are caused by some external unit or processor and are generally routed to a TP through the interrupt unit. Examples of this type include inter-processor communication (i.e. SLEEP and WAKE commands), I/O interrupts, etc.

The purpose of this section is to describe the operation of the TP when it is confronted with one of these two types of interrupts. In general it must record the particular type of interrupt which has occurred and any pertinent data which might be needed to handle it. The TP must then save its present status so that the current process can be resumed at a later time, and finally must transfer control to the system Interrupt Handler.



Interrupt Sequence Flow Chart

#### 4.6.1 Local Interrupts

Local interrupts are generated within a processor and are handled within the processor in which the interrupt condition occurs. Examples of this type of interrupt are: illegal operation, arithmetic overflow, illegal access, segment or page missing, etc. (see figure 4.6.1 for a full listing). The IU does not handle this type of interrupt.



TP Interrupts:

MNT	-	Page Map Not There	}	for Segment and Segment Name Table
PNT	-	Page Not There		
DNT	-	Data Not There		
TRP	-	Trap		
BOV	-	Bounds Overflow		
ILAC	-	Illegal Access		
PAR	-	Parity Error		
SUF	-	Stack Underflow		
ILI	-	Illegal Instruction		
AOV	-	Adder Overflow		
ASE	-	Available		
ILEX	-	Illegal EXIT		
PVV	-	Privilege Violation		

AU Interrupts:

OV - Overflow  
UN - Underflow (FLT)  
LS - Loss of Significance (FLT)  
ID - Invalid Decimal (BCD)

PAU Interrupts:

PAUI - PAU Interrupt  
(type of interrupt, i.e. bogus result  
will be determined by Supervisor from  
a halfword of data pushed into the OS)

EN Interrupts:

ILAD - Illegal Address  
PAR - Parity Error

Figure 4.6.1 Local Interrupts

#### 4.6.1.1 Local Interrupt Design Philosophy

The Taxicrinic Processor has one major complication with regard to local interrupts which is not found in most CPU's, namely, that many instructions perform irrevokable changes on the process data base before the existence of a local interrupt (or trap) is determined. Thus, if an interrupt does occur, it is not possible to merely stop the execution of the instruction and after the interrupt has been satisfied to begin at the beginning. Instead the state of the machine at some mid-point in the instruction must be saved so that the instruction can continue where it left off.

In the general case, this situation would require hundreds of bits simply to designate the sequence path which the instruction had taken at the time of the interrupt. Luckily, however, even though the instruction as a whole may not be restartable, there are many sequences which can be restarted if a local interrupt occurs within them. At the same time, since the TP has some flexibility in deciding when to recognize distal interrupts, it can always avoid looking for them during sequences which are "restartable". As a result, the number of possible "restarting points" can be greatly reduced since the TP will never have to return from an interrupt to any of the middle points of a restartable sequence.

Thus, in the design of the TP sequences themselves, careful attention is paid to the sequences in which it is possible for a local interrupt to occur. For every such situation, every effort is made to either 1) classify the interrupt as catastrophic (i.e. there will be no return to complete the instruction) or 2) write the sequence in such a manner that the interrupt will be detected before any permanent changes are made to the memory or the pointer or base registers by that sequence. If this can be done, then whenever a local interrupt is detected, a special "interrupt exit" can be made back to the calling sequence. This means that the sequence which detected the interrupt can be restarted when operation is resumed after the interrupt handling.

By going through this process at higher and higher levels of nested calls, the interrupt can be bucked back until eventually a return is made to a sequence which cannot undo a previous operation. Such a point in a sequence is called an Interrupt Point. After the interrupt has been processed, control will have to be returned to the proper Interrupt Point in order for the interrupted instruction to continue. In addition, these Interrupt Points are the only points in the control sequences where the TP looks for distal interrupts. The end result of this whole process is to reduce the number of possible machine states which must be saved in order to be able to restart after an interrupt.

The overall operation of the TP Local Interrupt system can be described as follows: as soon as an interrupt condition is discovered during the operation of some subsequence, the normal sequence halts, sets the proper interrupt indicator bits and after making any necessary adjustments makes a special interrupt return. The sequence which called the interrupted sequence will in turn, perform any operations which may be necessary to restore its status to a stable condition and then it will make a special interrupt return. This process continues until eventually a return is made to a sequence which cannot undo a previous operation, i.e. an Interrupt Point.

Once an interrupt has been detected and control has reached an Interrupt Point, the hardware transfers control to the hardware Interrupt Control Sequence. This sequence has the responsibility for determining the level of the recognized interrupt and for storing the information necessary for processing the interrupt in the Interrupt Storage Segment. In addition, it must save the necessary TP registers and status information which will be needed to restart the TP when processing resumes after the interrupt has been taken care of.

The information stored by the TP consists of two types: information needed to process the interrupt, and information used to restore the status of the TP when the interrupt processing has been completed. In order to keep the volume of the latter type of information as small as possible, only those registers are saved which are absolutely needed to specify the interrupt. If more registers are needed later on in interrupt processing, the procedure performing the processing has the responsibility for saving their contents. As a result, among the Pointer

Registers for example, only PR#0 and PR#1 are initially saved in the storage block. PR#0 must be used in transferring control to the Interrupt Handler, while PR#1 is used to point to the storage block itself.

In addition to holding the two types of information just mentioned, the Interrupt Storage Block is also used as temporary storage by the Interrupt Handler. This is necessary since the Interrupt Handler Procedure cannot necessarily assume that any data storage other than the Storage Block itself is still usable (i.e. Available Space may be exhausted, the Operand Stack may have had a bounds overflow, etc.). The TP will load one predetermined part of this area of the Storage Block with the entry which was used in the pointer circular buffer. This allows the Interrupt Handler to replace the original pointer entry with a new pointer to a new unused block of storage.

Once these storage operations have been completed, the TP has finished the operations necessary for the recognized interrupt.

It is possible, of course, that more than one interrupt was present at the time the TP recognized the interrupt situation. In such cases the TP chooses the highest priority interrupt, or if there is more than one at the highest priority it will choose one of them by some deterministic method. All other interrupts will remain pending.

After having stored the necessary information, the TP turns on the interrupt masks and transfers control to the appropriate processing level of the Interrupt Handler procedure. The transfer of control is accomplished by loading PR#0 with the standardized segment name for the Interrupt Handler procedure and the virtual address of the appropriate entry in the Interrupt Handler's branch table. This location will contain a branch instruction to the procedure which takes care of interrupts at the given level.

At the completion of the interrupt processing, the Interrupt Return sequence reloads the necessary TP registers, the control flip-flops and sets up the interrupt return state of the machine. Then it returns to the interrupt return point.

The following sections will give a more detailed description of the Interrupt Sequence and Interrupt Return Sequence.



#### 4.6.1.2 Interrupt Storage Segment

Upon detection of an interrupt, it is the responsibility of a Taxicrinic Processor to store certain registers, status flip-flops, and other indications of its current state along with enough information to tell the Interrupt Handler what type of interrupt took place. The purpose of this section is to describe the Interrupt Storage Segment (which contains space for this information), and to explain how information is loaded into and out of this storage area.

The Interrupt Storage Segment is unique for each task in the Illiac III system. Although the length of this segment can vary from one task to the next, depending on the expected amount and type of interrupt activity, the basic format is the same. As shown in Figure 4.6.1.2/1, the Interrupt Storage Segment is divided into three basic areas: the status area, the pointer circular buffer area and the storage block area.

The status area is divided into entries corresponding respectively to the various interrupt levels. Each entry consists of a Last Filled Block control double word (CDW) which indicates the status of the pointer circular buffer for the corresponding interrupt level. It indicates the pointer in the circular buffer which points to the next block to be filled with interrupt information at that interrupt level. At the end of the status area there is, in addition, an available space format entry which is used by the Interrupt Handler in assigning storage clocks to the various circular buffers.

The pointer circular buffer area is also divided into sections corresponding to the interrupt levels. Each section contains one circular buffer of pointer entries. These buffers may contain anywhere from one entry on up.

The storage block area comprises the rest of the Interrupt Storage Segment and consists of storage blocks used to contain the information to be stored during interrupts. These blocks, which are controlled by the standard available space techniques, are shifted on to queues, processed, and eventually returned to available space.

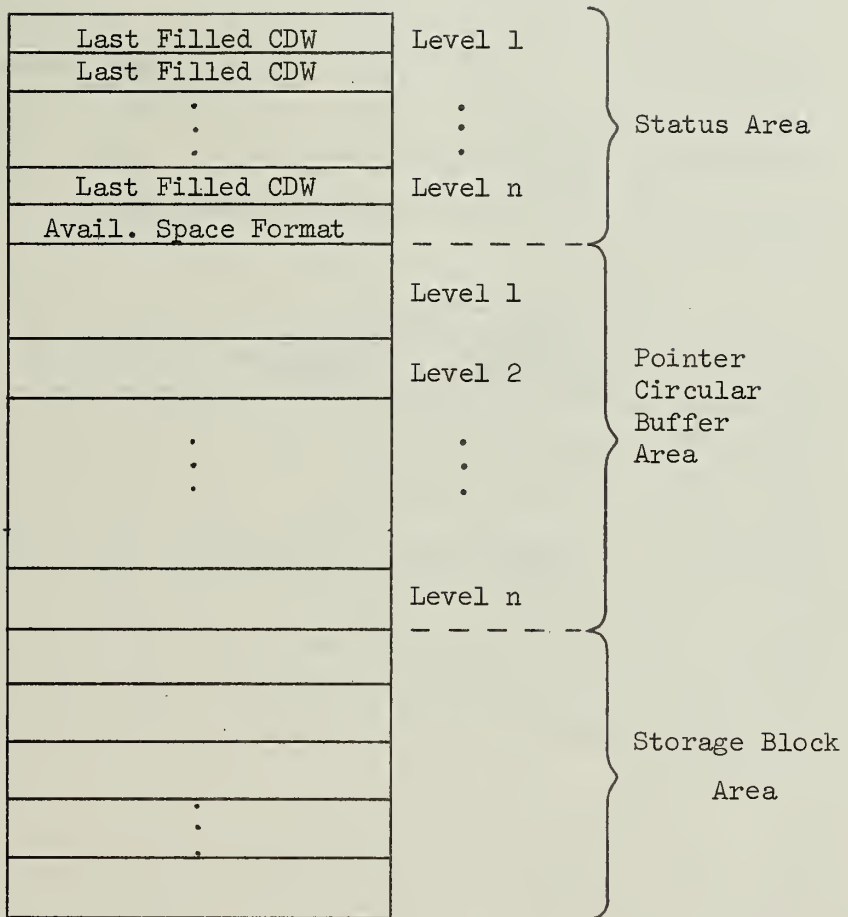


Figure 4.6.1.2/1 Interrupt Storage Segment Structure

As far as using the Interrupt Storage Segment is concerned, all storing of interrupt information is controlled through the control double words. The CDW's have the format shown in Figure 4.6.1.2/2. The rightmost halfword contains the pointer to the next entry in the corresponding level's circular buffer. The leftmost halfword field contains the address of the beginning of the circular buffer. The second halfword contains the increment field. The third halfword contains the beginning address of the first byte after the last entry in the corresponding level's circular buffer.

Given this format, the process of finding storage for the interrupt information at a given level interrupt is quite simple (refer to Figure 4.6.1.2/3). First an access is made to the appropriate last filled CDW.. The pointer thus obtained is then incremented by the length of an entry in the circular buffer (the increment field of the CDW). This new pointer value is used as the address of the pointer to the block into which the interrupt information is to be stored. If the new pointer value does not equal the maximum allowed address, it is stored in the pointer value field of the control doubleword. Otherwise, the initial value from the leftmost halfword field is stored in the pointer value field in the control word.

An important point to note is the case of the Supervisor which can in fact be running on several TP's at one time. In this case, or in general in the case of any task which may run on more than one TP at a time, a problem of processor interference during the interrupts occurs. If two "Supervisor" TP's (i.e. TP's on which the Supervisor is running) both have interrupts at the same level at the same time, they could end up filling the same storage block in the Interrupt Storage Segment. To avoid this situation, it is necessary for the control word manipulations to be performed during only one Exchange Net access. Since the Exchange Net does not allow more than one TP to access a given core box at one time this restriction will ensure that by the time the second TP gets to the Last Filled Block control double word, it will have been updated to reflect the use of one of the storage blocks by the first TP. The only operation



Initial Value	Increment	Maximum Value	Pointer Value
------------------	-----------	------------------	------------------

Figure 4.6.1.2/2 Control Double Word Format

CDW:

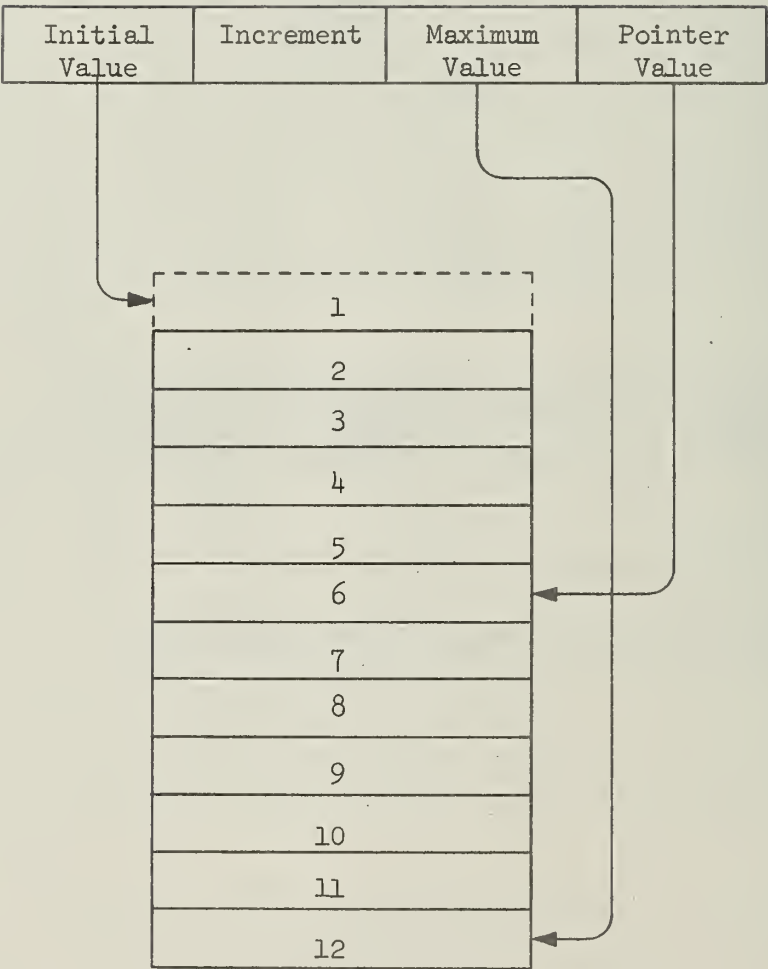


Figure 4.6.1.2/3 Use of CDW to Obtain an Interrupt Storage Block

necessary on the part of the first TP is that it must not let go of the Exchange Net until after it has updated the pointer entry and replaced it. This is done using the logic of the Increment and Check (INCK) instruction.

### 4.6.1.3 Interrupt Sequence

#### 4.6.1.3.1 Interrupt Sequence Description

The purpose of the Interrupt Sequence is to store the Interrupt Status of the TP in the Interrupt Storage block and to prepare the TP for a transfer. In order to load the storage block it is necessary to make several memory accesses, first of all to determine which block to fill and then to actually fill the block. However, since the original interrupt may occur in the middle of a memory access or at some other point where all of the registers contain data which must be saved, a method must be found for saving this data while making accesses to determine the interrupt storage block to be used. The means which was decided on was an Interrupt Storage Memory internal to the TP.

The IC interrupt storage memory is made up of 16, 36-bit words of memory. It is arranged on 8 cards in such a manner that it can collect the interrupt information simply by pulsing the proper counter at the proper rate. Once this information is stored, the TP can then make use of the registers which originally contained that information to access memory, find the interrupt block and then read out the saved interrupt information from the IC interrupt storage memory into the core memory interrupt storage block.

The Interrupt Sequence itself is reasonably straightforward. Referring to the flow chart at the end of this section, it can be seen that the initial steps consist of loading the Interrupt Storage Memory with the contents of various registers. Note that the zeroth entry of the interrupt storage has already been filled by the sequence which detected the interrupt. The order in which the registers are stored is fixed so that those registers which are needed to merge and build up the more complicated data (i.e. the LR and DR) are saved first. Note also that the IR is not stored until after the miscellaneous data bits. This is necessary because on an interrupt return the IR will be used to restore the TGR and mnemonic byte data and can thus not be loaded with its restored information until the completion of this operation.

Once the interrupt storage memory is loaded, the next task is to determine the address of the Interrupt Storage Block to be used

to contain this interrupt information. Figure 4.6.1.2/1 depicts the structure of the Interrupt Storage Segment while figure 4.6.1.2/3 shows a circular buffer of interrupt block entries. In order to access the Interrupt Storage Block the control sequence first loads PR Segment Name Register #0 with zero, i.e. the segment name of the Interrupt Storage Segment and the DR with the number of the interrupt level multiplied by 8. This latter quantity is used to determine the address of the control double word for the interrupt level of the current interrupt. During the loading of PRSNR(0), the PR Segment Name Selector Register is set to select PRSNR(0) so that when the Increment and Check sequence is entered, it will process the control DW within the interrupt storage segment corresponding to the current interrupt level. Note that prior to loading PRSNR(0) with the segment name of the interrupt segment (i.e. "0"), PRSNR(1) is loaded with the same name. This is necessary since PR#1 will be needed to access the interrupt block once control is passed to the Interrupt Handler Procedure.

When the Increment and Check sequence returns, the DR will contain the virtual address of the entry in the circular buffer for the interrupt level of the current interrupt. The sequence accesses this halfword entry to obtain the virtual address of the interrupt storage block. This address is stored in PR#1 and the DR and then the Modified Entry of the memory sequence is used to calculate and check the 24-bit actual address which eventually is put in the AR, right justified.

Since each access to the Interrupt storage block will be on the same page, the calculation of the core address and all of the validity checks need only be performed once. Thus we can save a great deal of time by performing the address calculation before entering the memory storage access loop.

Once control reaches the memory storage access loop, successive words from the IC interrupt storage memory are loaded into the LR and DR

and then written into the core memory. The format for the interrupt storage block showing the location of the various interrupt information is given in figure 4.6.1.3.1/1.

Note that a check is made at every point in the sequence where an interrupt might be detected due to memory or exchange net failure. If at any time such an interrupt occurs, disaster has struck and about the only thing which can be done is to turn on a light and start some bells ringing.

0	BLOCK POINTER		FOR USE BY SYSTEM	
8	SEGMENT NAME		VIRTUAL ADD.	LR
16	DR			PR# 0
24	PR# 1			PRSNR# 0      PRSNR# 1
32	SBR			AR
40	TGR		INST. MNEM.	CONTROL FF STATUS
48	CALLING CONTROL POINT STATUS			
56	PRIORITY LEVEL	INTERRUPT TYPE		MISC. STATUS
64	IR			

Figure 4.6.1.3.1/1 Interrupt Storage Block Format



#### 4.6.1.3.2 Interrupt Status Collection Logic

The Interrupt Status Collection Logic is the logic used to gather up the data which is to be stored in the IC Interrupt Storage Memory and to gate it to the input lines of this memory. In general, there are two levels of saving which might be involved in an "interrupt" (in this case using the word in a very broad sense). the higher level there is the TP status necessary to restart the interrupted task. At the lower level there is the TP status which is necessary to restart the interrupted instruction. The distinction between these two levels will now be explained.

In general there is quite a bit of information which must be stored when a task is removed from a TP. This basic information is stored in the Task Control Block when the task is inactive. Although it is true that many interrupts will eventually result in having a given task taken off of the TP, this is not always true. In these latter cases it would be quite inefficient to store all of these quantities if they are not going to be used. Therefore it was decided that the interrupt sequence itself would only save that information which was necessary to describe the interrupt and to restart the particular instruction which was interrupted. If it later becomes obvious that the task is going to have to be taken off the TP, then the Supervisor is responsible for saving the rest of the task information in the TCB.

On an interrupt return the procedure will be reversed. If the task has been inactivated part of the job of the supervisor reactivation procedure (see Section 5.6.5.2) will be to reload the necessary registers from the TCB before returning control to the task.

In order to simplify the process of collecting the interrupt status bits and registers which must be saved by the hardware interrupt sequence, each input bit line to the interrupt storage memory of the TP has been attached to an IC multiplex chip, as shown in figure 4.6.1.3. Thus by hooking the various input lines of the multiplex chip to the various status bits, the status bits can be gated to various words in the interrupt storage memory, by counting up on the multiplexor and memory address counters simultaneously.

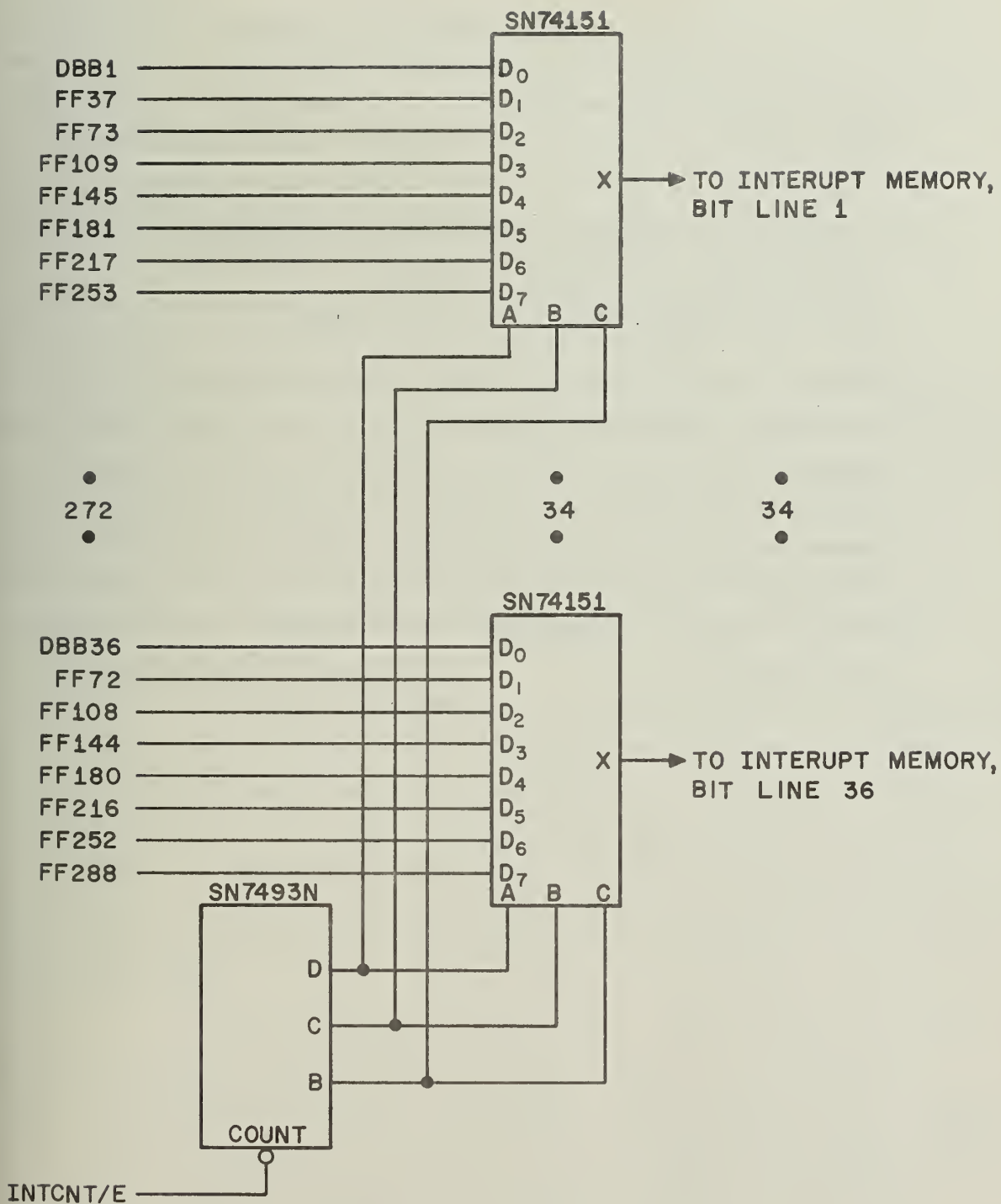


Figure 4.6.1.3.2/1 Interrupt Status Collection Logic

Note that the zeroth input line of each multiplex chip is attached to the DBB bus. This allows registers to be gated into the interrupt storage memory through the Permuter. Note that since the memory address counter and the multiplex counter are controlled independently, an arbitrary number of full word registers can be stored in the interrupt storage memory before using the multiplexor counter to store the various miscellaneous bits.

The miscellaneous bits being stored consist of status flip-flops, small registers such as the TGR, several types of interrupt information such as priority level, type of interrupt, etc., and the status of those calling control points which may be active before entering the Interrupt Sequence. This latter information must be saved in order to be able to determine the proper return paths when processing resumes after the interrupt has been handled. It is the calling control point which indicates which particular sequence made the call when it is time for the called sequence to return control (see Appendix 4.A).

#### 4.6.1.4 Increment and Check Sequence

##### 4.6.1.4.1 Increment and Check Sequence Description

The Increment and Check Sequence is used to update control double words of the format shown in figure 4.6.1.4.1 below:

Initial Value	Increment Value	Maximum Value	Pointer Value
------------------	--------------------	------------------	------------------

Figure 4.6.1.4.1 Control Double Word Format

The sequence accesses the control double word at the virtual address which was previously stored in the DR. The PR Segment Name Selection Register must also have been set before entering the sequence. When the data returns from the memory unit the sequence increments the pointer value field by the contents of the increment value field without releasing the Exchange Net. This insures that no other TP will be able to access the control double word until the sequence has completed its operations.

When the incrementation has been completed, the sequence checks the incremented value against the value in the maximum value field. If the incremented value does not equal the maximum allowed address, it is stored back in the pointer value field of the control double word. Otherwise, the initial value field is loaded into the pointer value field. In either case the original pointer value field is loaded into the DR right justified and the EQ flip flop is set according to the status of the test.

This sequence has several uses. First of all, it is used for controlling the Interrupt Storage Segment circular buffers. In this case, the initial and maximum value fields denote the beginning and end of the circular buffers and the increment value field is set to two, i.e. the length of the halfword entries in the circular buffer. In the general case, when used as part of the INCK instruction, the sequence can be used to control arbitrary buffers using the same general format.



Secondly, the sequence (by means of the INCK instruction) can be used in controlling "critical sections" so that only one programmer will utilize a section of core at a time. In this case, a control double word is set up for each such "critical section". To be used in this manner, the pointer value field of the double word is originally set to one, the increment field is zero, the initial field is 0 and the maximum field is 1.

When a processor wants to use a critical section, it first performs an INCK on the control word for that section. If there is no processor in the section, the pointer value will be 1 and after being incremented by 0 it will equal the maximum value field, thus causing the initial value of 0 to be returned to the field. Then after testing EQ and discovering it is on, the TP can begin processing in the critical section.

Now suppose a second TP wants to process in this same area before the first TP is finished. In this case it will also perform an INCK. This time, however, the new pointer value, after being incremented by 0, will still be zero and will thus not be equal to the maximum value field of 1. When this second TP checks the EQ indicator it will be off and thus the second TP will have to "go to sleep" for awhile.

Eventually the first TP will finish its processing and at this point it will set the control word pointer value back to 1. Thus eventually when the second TP again checks the control word, it will be allowed to use the "critical section".

Note that the biggest problem with this procedure is that it **relays** on the "honesty" of a programmer to check the appropriate entry before using a "critical section" and also to reset the entry when it is finished.

A third use of the sequence is as a means of controlling loops. However, this will only work if it can be guaranteed that the pointer value will eventually equal the maximum value exactly.

A flowchart for the Increment and Check Sequence is given at the end of this section.



#### 4.6.1.5 Interrupt Return Sequence

##### 4.6.1.5.1 Interrupt Return Sequence Description

The purpose of the Interrupt Return Sequence is to restore the status of the TP to the previous state indicated by the Interrupt Storage Block whose initial virtual address is given in the DR. The PR Segment Name Selector Register must be set before entering the sequence. The sequence also returns the Interrupt Storage Block to the Available Space area.

The operations involved are shown in the flow chart at the end of this section and are approximately the reverse of the operations performed by the Interrupt Sequence (section 4.6.1.3.1). The first task is to load up the Interrupt Storage Memory from the Interrupt Storage Block. Then the Interrupt Storage Block is placed back on the Available Space list using the core memory version of AS RESTORE.

The next task is to restore the status of the various registers. The restoration of the LR, DR, PR(0), PR(1), SBR and AR are very straightforward. In the case of PRSNR(0) and PRSNR(1), each register is only 2 bytes long so that the output from the corresponding interrupt storage memory word is gated out twice and permuted in the case of PRSNR(1).

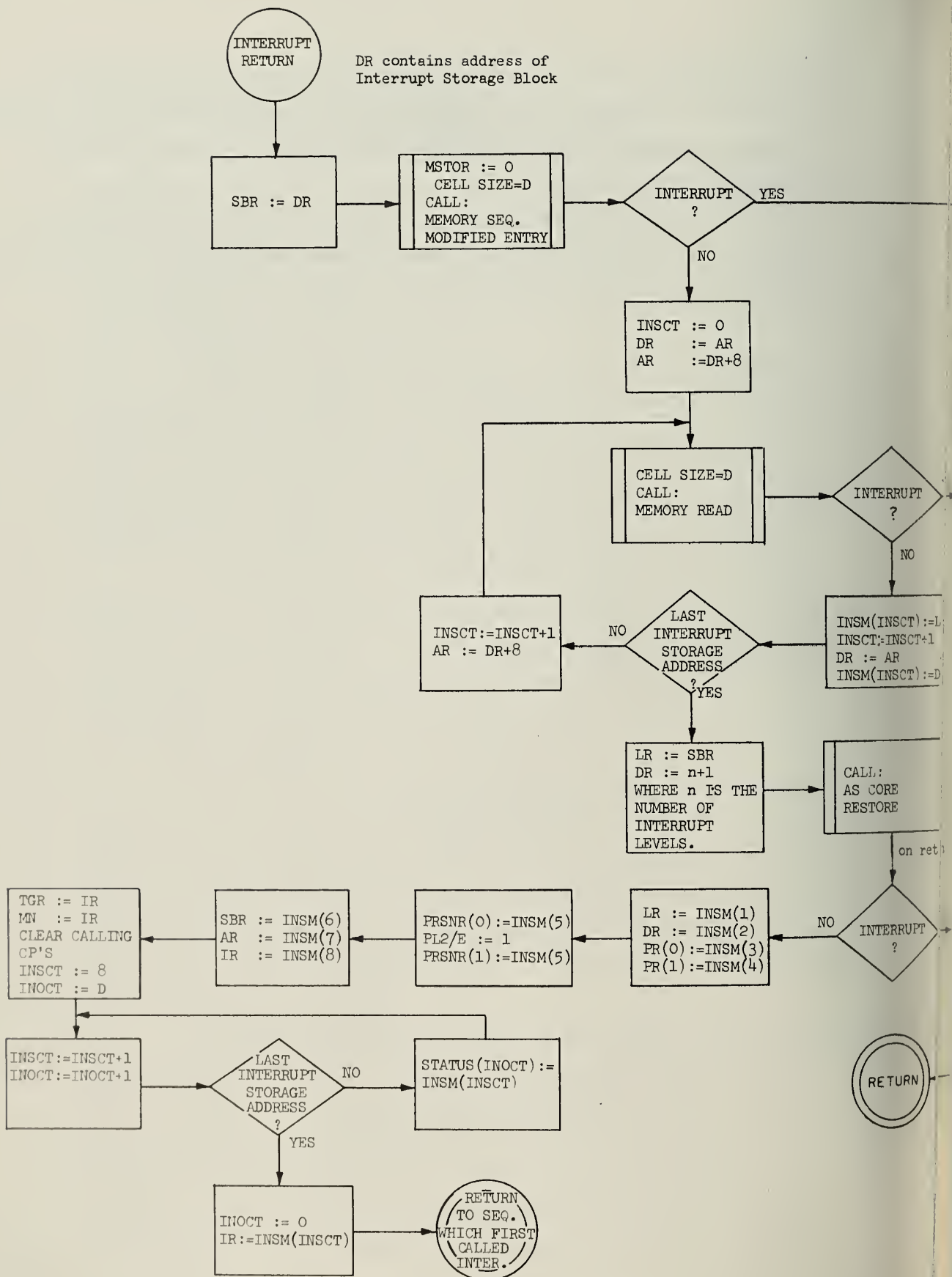
The eighth word in the interrupt memory contains the original contents of the TGR and the mnemonic register. These quantities are gated into the IR from which position the respective registers can be loaded in the conventional manner. The remaining status bits are then restored using a counter setup similar to the multiplexing on the input to the interrupt storage memory. This logic is described in detail in section 4.6.1.5.2.

Note that before loading the status bits, all of the calling control points are reset. Then when the calling control point status is read out, these control points are set according to the status bits which were saved when the interrupt was detected. In particular the calling control point which first called the Interrupt Sequence at the original interrupt point will be set. This means that when the Interrupt Return sequence has completed its operations, it can return to the



proper interrupt point simply by returning as if it were the end of the Interrupt Sequence. Thus every calling control point which calls the Interrupt Sequence will have its return signal line coming from the Interrupt Return sequence.

The last operation in the Interrupt Return sequence is to restore the status of the IR. Then the sequence returns to the Interrupt Point to take up operations where it left off previously.



INTERRUPT RETURN Sequence Flow Chart

#### 4.6.1.5.2 Interrupt Status Restoration Logic

The Interrupt Status Restoration Logic is used to restore the status of the various control flip-flops, calling control points and small registers which were saved in the interrupt storage block. This data is first gated to the Interrupt Storage Memory from which point the individual words can be gated to the Permuter. The output of the permuter can then be sent to the status restoration logic shown in figure 4.6.1.5.2 and from there is sent to its respective positions.

The routing of the data is controlled by a set of interrupt output counters which control the demultiplexing operation. When set to zero all outputs will be off since the zero output is not connected. When it is necessary to restore the status, the corresponding words from the Interrupt Storage Memory are gated to the permuter and the interrupt output counter is counted up so that the data is sent to the proper place. Note that the inputs to the input multiplex chip in figure 4.6.1.5.2 do not correspond to the same outputs in figure 4.6.1.5.2. This is due to the fact that the first word of "miscellaneous status bits" has already been gated to its proper position using the IR gating features. Thus, that word need not be gated out again.

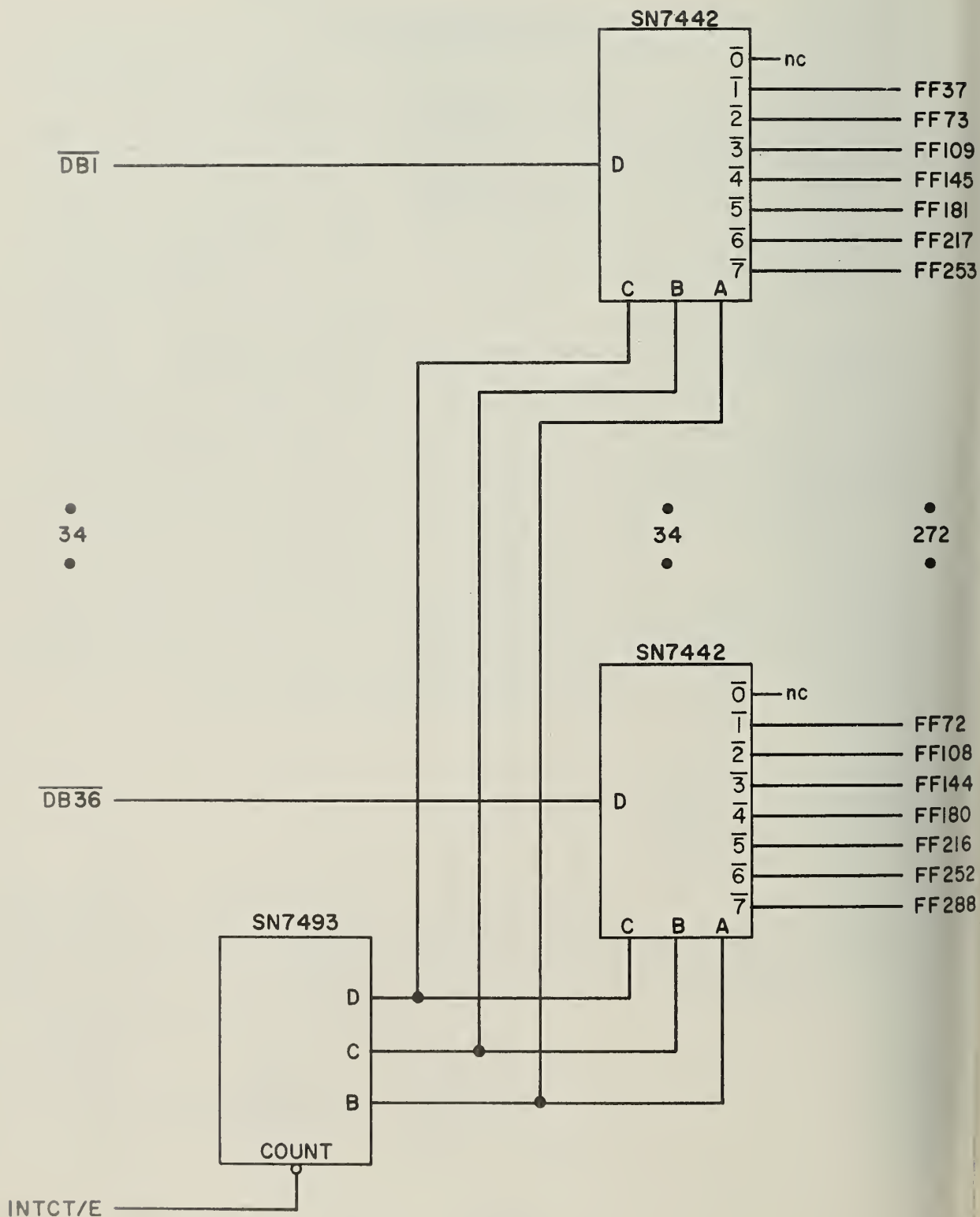


Figure 4.6.1.5.2 Interrupt Status Restoration

7/15/71

#### 4.6.1.6 AS Core GET Sequence

##### 4.6.1.6.1 AS Core GET Sequence Description

This control sequence is an adaptation of the AS GET sequence which obtains its AS control element from the memory rather than from a Pointer Register. The purpose of this sequence (and the instruction which is based on it) is to allow the programmer to store his available space control elements permanently in memory. This avoids the necessity of continually shifting them back and forth to a single PR or of having to use up several PR's as available space file control elements.

Having the AS control elements in core presents an additional problem which is not present if they remain in PR's, namely the problem of conflicts between 2 or more processors trying to use the same element. In order to solve this problem, the sequence loads the link portion of an element with all 1's whenever it begins manipulation on the AS file controlled by a particular element. It also checks before it begins its manipulation, to see if the link is already all 1's. This allows it to go into a sleep state if some other processor is using the file and avoids processor conflicts.

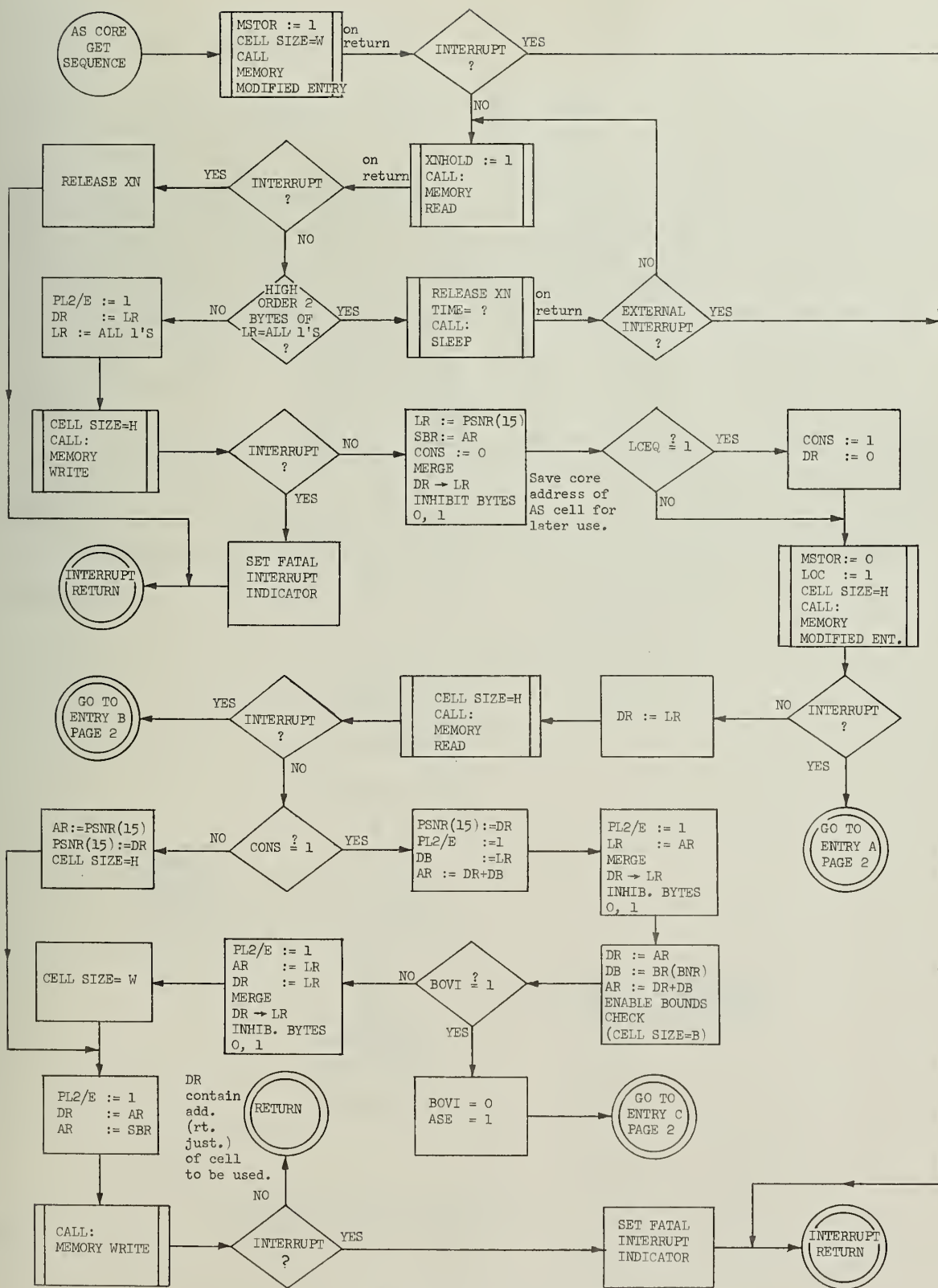
Other than this the sequence is reasonably straightforward and follows the same general pattern of the AS GET sequence (see Section 4.3.1.2). After reading out the AS control element link and count field and resetting the link field to all 1's the sequence determines which cell will be the new cell to be loaded into the link field of the AS element. Before doing this, however, it saves the core address of the AS control element in the SBR so that it will not have to repeat the address construction phase of the memory access sequence when it reloads the AS control element.

The new link field of the AS element will either be the address of the next cell on the free list after the current top cell has been removed, or it will be the next cell in the continuous storage area after a cell has been removed from it. In either case a memory access must be made, either to the link field of the current top cell on the free list, or to cell zero of the available space file to determine the length of the cell to be taken off of contiguous storage. Once this has been determined,

it is loaded back into the AS control element.

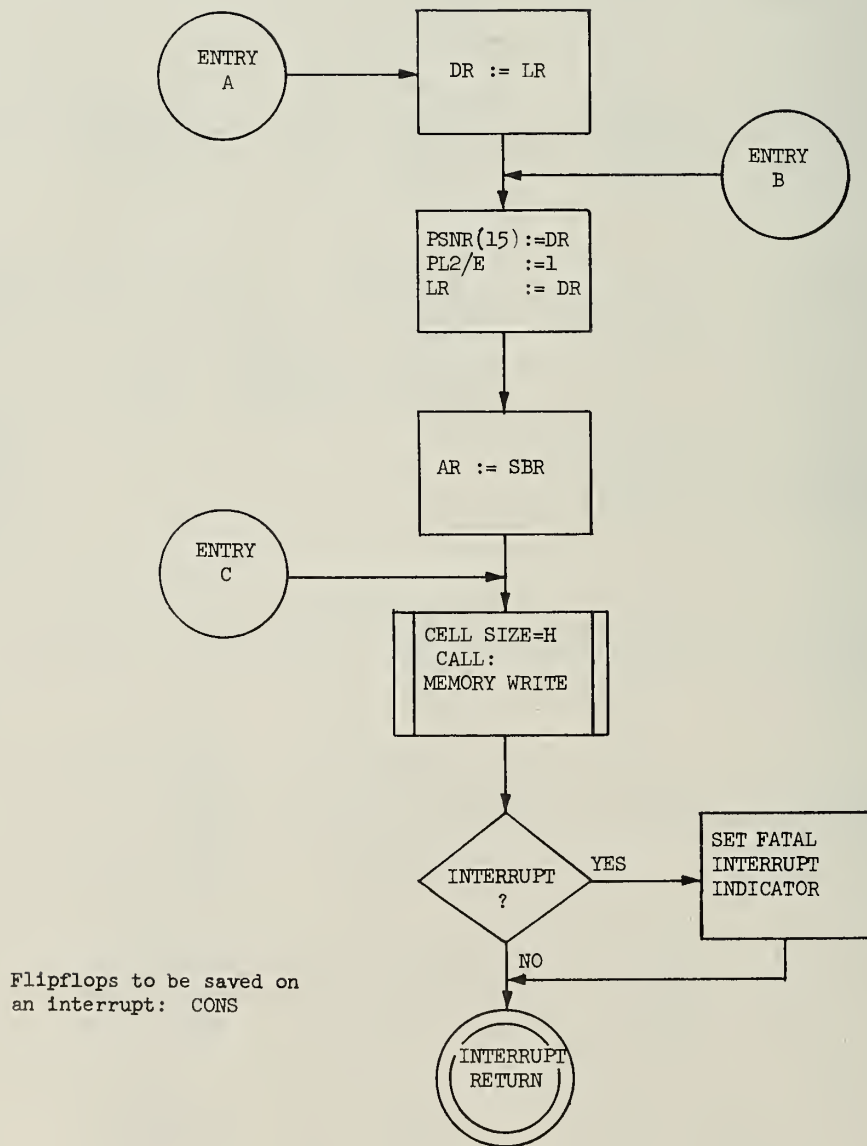
Note that the interrupt situation is somewhat complicated. In some situations it is possible to recover while in others if an interrupt occurs it is quite possible that the AS control element will be left in a permanent lock out. Such an error is obviously very fatal. Hopefully however these will not occur in normal operation and the only time they should occur is if the memory unit fails between the time the AS control element is first locked out and the time the sequence tries to unlock it.





AS CORE GET Sequence Flow Chart





AS CORE GET Sequence Flow Chart

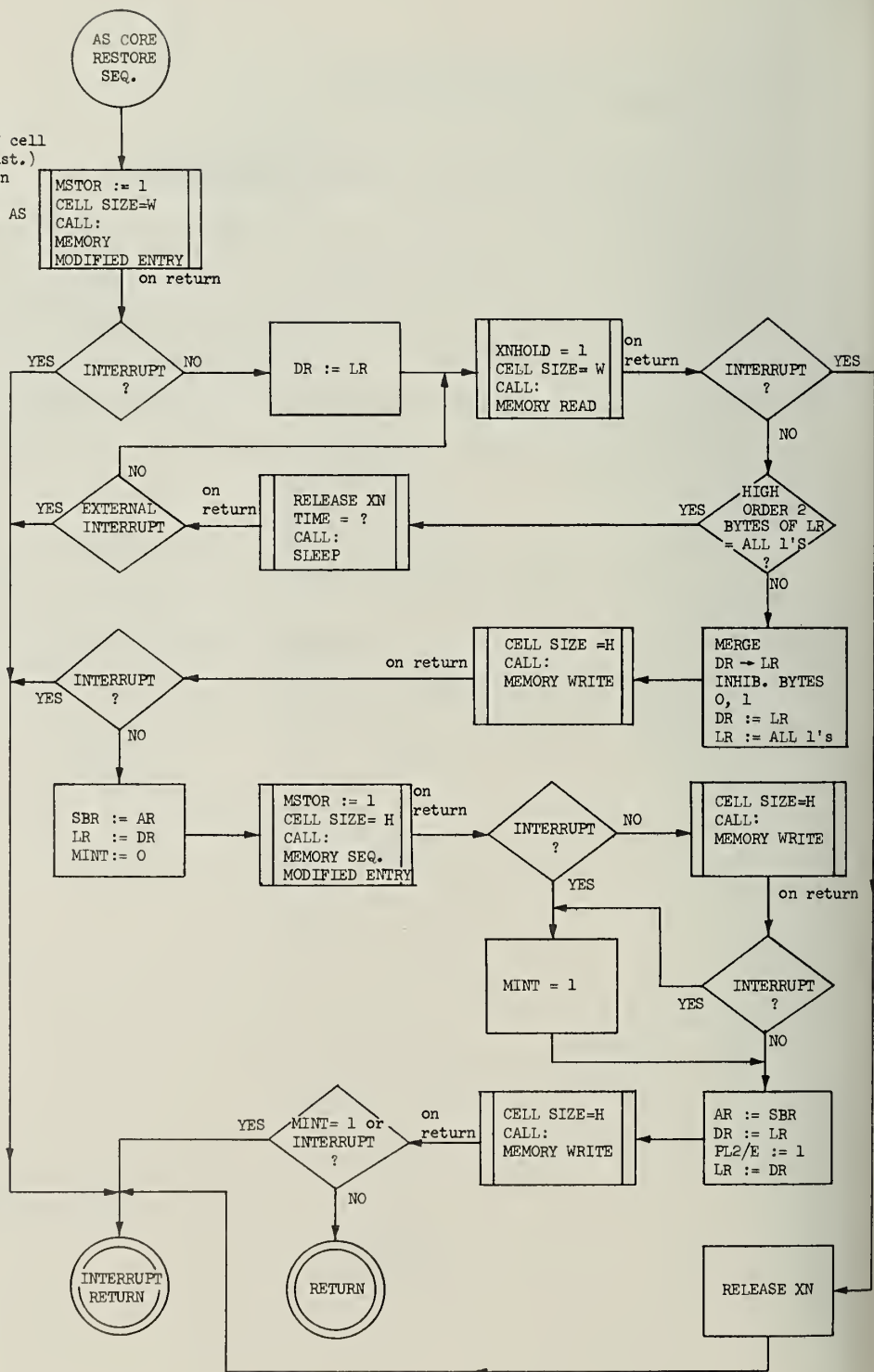
#### 4.6.1.7 AS Core RESTORE Sequence

##### 4.6.1.7.1 AS Core RESTORE Sequence Description

This control sequence is an adaptation of the AS RESTORE sequence which obtains its AS control element from the memory rather than from a Pointer Register within the TP. The purpose of this sequence (and the instruction which is based on it) is to allow the programmer to use core as the storage area for his Available Space control elements and to use them from there directly rather than first having to load them into Pointer Register.

The problems involved in this modification are described in section 4.6.1.6.1 in the discussion on the AS Core GET sequence. The AS Core RESTORE sequence is perfectly straightforward, given the solutions to the these problems as described in Section 4.6.1.6.1, and follows the general outline of the AS RESTORE sequence (see Section 4.3.1.3).

LR contains address of cell  
to be returned (rt. just.)  
PRSNR selector has been  
set.  
DR contains address of AS  
cell. (rt. just.)



AS CORE RESTORE Sequence Flow Chart

#### 4.7 The Display Console and Manual Intervention

The design of the Taxicrinic Processors provides for the use of an Engineering Console capable of displaying the status of various registers in the several TP's. In addition the console allows the maintenance engineer to manually intervene in the operation of the processors and change the contents of various registers, if he so desires. This console can also be used to display information pertinent to the IOP's and the other Units of the Illiac III system.

The purpose of this section is to describe the overall design philosophy used in designing the TP/Engineering Console interface and then to describe in detail both the interface itself and its interaction with the design of the TP logic.

#### 4.7.1 General Philosophy

Each Taxicrinic Processor is connected to the Engineering Console via the standard 50 line Exchange Net interface over which commands and data are sent. There are 36 data lines which encompass four bytes of data. The remaining lines are control lines. The data lines operate in the simplex mode, i.e. data can only be transmitted in one direction at a time. One control line is used by the TP to tell the Engineering Console that the required operation has been performed. The remaining control lines are used by the console to tell the TP what command to execute.

Basically the tasks of the Engineering Console are twofold: first to maintain a display of one or more of the processors and/or units attached to it and, second, to allow the maintenance personnel to manipulate the internal state of these devices and perform various diagnostic tests. In order to perform these tasks the Engineering Console issues various commands to the processors and/or units requesting either items of information or instructing them to perform certain actions. The console contains an indeterminate amount of storage which it updates as required and then displays.

When a given processor or unit is running there is little point in having the console try to keep its display current since the operator would only see a blur anyway. Thus a basic convention has been established that whenever a processor or unit is running, the only commands from the console which it will obey are a small subset telling it to halt. In the case of the TP these are the Set Instruction Halt at next instruction and console interrupt commands (see Section 4.7.2.1 and 4.7.2.8 respectively). This restriction greatly simplifies the logic in the processors and units connected to the console since it ensures that aside from these commands the console will not interfere with normal running operations.

The basic idea used in designing the console itself and the logic within the various processors and/or units is that the Engineering Console will establish its own goals on the basis of instructions given it by the maintenance engineer (e.g. display the contents of all the Pointer Registers in TP#1) and will accomplish these goals by issuing the necessary commands to the processors and/or units connected to it. The connected devices need only obey the commands as they arrive. This arrangement has several advantages: first, it places the overall control in one place and prevents duplication of this logic in each processor and unit attached to the console. Second, it allows the processor and unit design to be decoupled from the actual diagnostic operations to be performed by the Engineering Console. This in turn means that if the commands to the processors/units have been properly chosen, the duties of the console can be changed extensively without any modification to the processors/units attached to it.

As to the actual implementation of the console, it can range anywhere from an initial control panel of switches connected with the necessary control logic to issue the commands all the way to a mini-computer. This latter implementation appears preferable for several reasons:

- 1) it allows flexible communication (assuming the proper device commands have been implemented) and easy modification of command strings.
- 2) It converts the problem of designing the Engineering Console from a logical design problem into a programming problem.
- 3) This strategy will be cheaper in the long run given the present price structure for mini-computers.



- 4) The computer's memory would obviously be available for storing register contents. But in addition to this it can be used to store various decision tables which would allow the maintenance engineer to enter mnemonic codes when referring to sequence names (which in the case of the TP are needed in the Execute Sequence and the Set Sequence Halt commands) instead of having to look up numbers in the tables himself and entering them through switches. Of course, table-driven diagnostic sequences can be executed.
- 5) The console computer is more amenable to the implementation of long strings of console commands. At present it is thought that this might be one way of setting up the computer so that the system supervisor can be loaded. Obviously it is desirable that this sequence be flexible enough to allow for the variability of the operating system from one week to the next.



#### 4.7.2 Engineering Console Commands to the TP

The purpose of this section is to list the various console commands which can be issued by the Engineering Console to a Taxicrinic Processor. The operations entailed by each command are listed in figure 4.7.2 and are explained in detail in the following sections.

<u>COMMAND</u>	<u>COMMAND</u> <u>FIELD</u>	<u>OPERAND</u> <u>FIELD</u>
Set Instruction Halt	7	10
Reset Instruction Halt	7	11
Set Maintenance Halt	7	12
Reset Maintenance Halt	7	13
Set Sequence Halt	1	--
Reset Sequence Halt	7	31
Execute Sequence	2	--
Interrupt	7	20
Interrupt Return	7	21
Run	7	22
Load Register	3	--
Read Register	4	--
Auto Load	7	23
Position	7	30

Figure 4.7.2 Console Commands to the TP.

#### 4.7.2.1 Set Instruction Halt

SIH causes the indicated TP to cease operation as soon as it completes the current instruction and attempts to start execution of the Main Control Sequence. This operation is accomplished by setting the Instruction Sequence Halt flip-flop, IHLT, to 1. This will automatically inhibit the initiation of the Main Control Sequence when the present instruction cycle is finished. If an interrupt occurs during the present sequence, the halt will not occur until the TP attempts to use the Main Control Sequence. This will probably be at the first instruction in the interrupt handling procedure.

The SIH command is one of the two console commands which can be obeyed by a running TP. The other command is INTR, the Console-TP Interrupt command.

#### 4.7.2.2 Reset Instruction Halt

The RIH command resets the first stage of the Instruction Sequence Halt flip-flop, IHLT. The next time a run or interrupt return console command is given, the second stage of the IHLT flip-flop will be reset which will allow the TP logic to continue provided the other halt indicators are off.

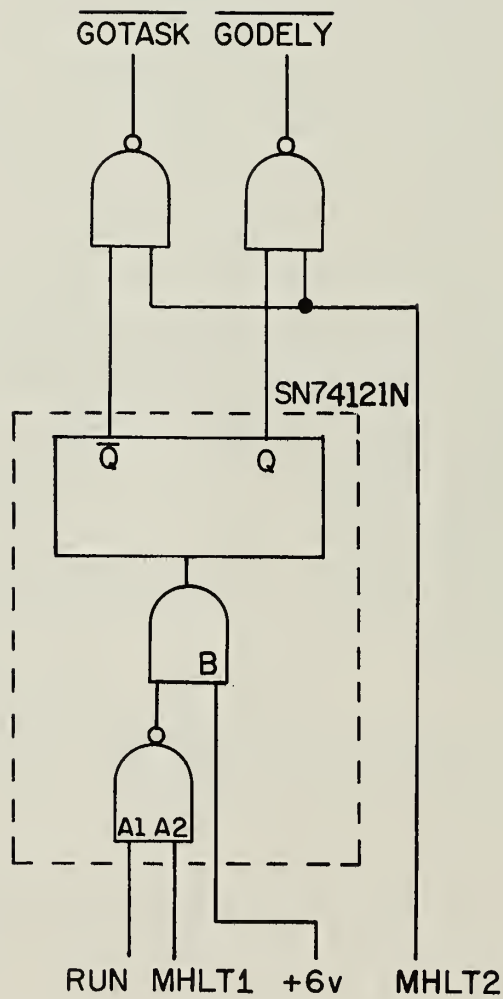


Figure 4.7.2.3/2 Logic for Production of GOTASK and GODELY Signals

If we were to try to use a single cycle method with only a GOTASK signal we could still stop the sequence at a given control point by turning off its GOTASK signal. However, in order to step to the next control point in the sequence we would either have to have separate GOTASK signals for each control point or we would have to turn on a common GOTASK signal for just enough time to have the task completed but not enough time to let the next control point get started. Although the latter possibility is far fetched, the former has several good points. Its major disadvantage is that it would require many more separate GOTASK signals and some method to decide which ones should be on and which should be off. Another advantage for this system is that it would allow skips of several control points if desired.

The final decision was made on the basis of the simplicity of the first design. However, at a later date some combination of the two might be preferable. One possibility might be to use the same numbers assigned to each control point within a sequence for the purpose of locating the currently active CP (see Section 4.7.2.3) as the identifying number for the purpose of a manual halt. This would involve a significant addition to the hardware however.

#### 4.7.2.4 Reset Maintenance Halt

The RMH command resets the first stage of the Maintenance Halt flip-flop, MHLT, to zero. The next time a run or interrupt return console command is given, the second stage of the MHLT flip-flop will be reset causing all of the maintenance stop signals to be turned off (i.e. GODELY and GOTASK will both be "1").

#### 4.7.2.5 Set Sequence Halt

The SSH command may only be issued when the TP is not running. It causes a bit to be set in the sequence halt selector corresponding to the sequence whose number is encoded in the 6 bit operand field in the command. The sequence halt selector then causes a signal to be sent to the control logic of the designated sequence which will inhibit the operation of that sequence if it is ever called by some other portion of the control logic. This inhibition is performed as shown in Figure 4.7.2.5.

The sequence selected will almost surely be processor dependent and thus this is a difficult command to interpret at the console. If it were the maintenance engineer's responsibility to enter the sequence number manually by setting switches, the implementation would be easy. However since he would be using several lists of sequence numbers, one for each processor or unit communicating with the console, this method would be highly prone to error. What is more, it may be difficult for him to detect that he has in fact made an error.

One solution is to implement the console using a mini-computer, as suggested above. A typewriter or teletype can be used for communication. The maintenance engineer then types in alphanumeric information instead of binary. Table lookup procedures can then be used to select the proper sequence number for the selected type of processor.



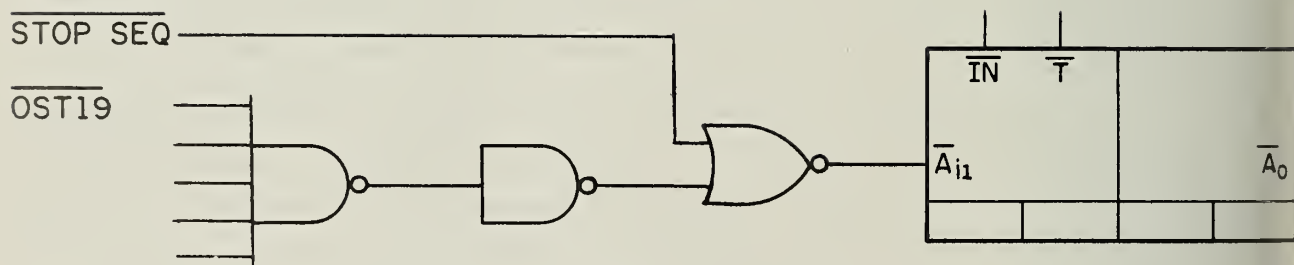


Figure 4.7.2.5 Technique for Halting Sequences

#### 4.7.2.6 Reset Sequence Halt

The RSH command turns off all of the bits in the sequence halt selector. This means that no sequence operations will be stopped in the future unless one of the selector bits is turned on by a subsequent SSH command.

#### 4.7.2.7 Execute Sequence

The EXEC command executes the sequence specified by the 6-bit operand field in the command. The last byte of the data field is used to supply the desired states of the control signals and flip-flops for the sequence to be executed. The table in Figure 4.7.2.7/1 gives the order of "parameters" for the various sequences. This field is decoded using the same logic as in the Set Sequence Halt command. In this case however, the decoded output is used to activate a calling control point to the selected sequence. When the sequence returns, the TP signals to the console that it has finished.

Number	Entry Point Name	PARAMETERS			
		1	2	3	4
0	Memory Direct Entry	MSTOR		XNHOLD	
1	Queue Counter Update				
2	Part. Mode Add. Conver.		SNAMT		
3	Memory Read			XNHOLD	
4	Memory Write		IMF	XNHOLD	
5	OS CLEAR				
6	OS ENTRY	POP	CS2C	DUP	
7	Overflow Entry				
8	SCR Modified	S/E	WAIT	X2	
9	OS Read	NEG	WAIT	NWTOS	STOR
10	OS Write		WAIT		
11	Memory Modified Entry	MSTOR		XNHOLD	
12	AS Get	ASTRS			
13	AS Restore	ASTRS			
14	Stack PR				
15	Unstack PR				

Figure 4.7.2.7/1 Parameter Positions for Execute Sequence Command  
for Sequences in the Core Machine

#### 4.7.2.8 Interrupt

The INT command causes the TP to react as if an interrupt of the highest priority has taken place. This console-generated interrupt is not maskable. Therefore it allows the console to take control of a TP independent of what else is happening in the environment. The command also causes the Instruction Halt flip-flop, IHLT, to be set so that the TP will halt as soon as it has stored all of the interrupt information and tries to execute a new instruction.

The INT command is one of two console commands which will be obeyed by a TP while it is running. The other is SIH, the Set Instruction Halt command.

The INT command can be used to save the state of an operating TP whenever an engineer wants to commandeer it to perform some diagnostic operation. Once this command has been issued the TP can be used in the sequence exercise mode without fear of destroying valuable information in the registers.

After an INT command has been issued the console can also be used to execute the interrupted program in a step-by-step manner. In this case the Interrupt Return command can be issued without resetting the IHLT and/or MHLT flip-flops, thus causing the TP to single step on an instruction basis or a maintenance halt basis. Eventually, if it is desired to restart the TP in exactly the same condition as prevailed when the interrupt occurred, it is only necessary to reset IHLT and MHLT and then issue an Interrupt Return command.

#### 4.7.2.9 Interrupt Return

The INTRN command causes the TP to restore its registers and control points to the state which prevailed at the time of the last Interrupt command. The program which was on the machine at that time will be reactivated and allowed to continue. However since IHLT and/or MHLT may possibly be still set, the program may stop soon after it has been reactivated. If these flip-flops are off, however, the program will continue in the normal fashion.

The Operating System has the power to restart a program which has been interrupted by a INT command, but only by issuing a subsequent INTRN command through the console.

#### 4.7.2.10 Run

This command allows TP operation to continue. It will reset the second stage of the instruction halt flip-flop or the maintenance halt flip-flop if either of these flip-flops have been given a reset command. The TP will again halt at the appropriate place if any of the halt indicators are still set. If all halting conditions have been reset, the program execution will continue in the normal fashion.



#### 4.7.2.11 Load/Read Registers

These LOAD and READ commands cause the designated register to either be loaded from the data lines or read out onto them. The desired register is indicated by a 6-bit field within the command. The register assignment is shown in Figure 4.7.2.10/1. Note that for consistency and ease in decoding, all storage which consists of blocks of registers uses the same bits for selecting which register is to be loaded or read out.

These operations work by utilizing the TP Permuter logic to access the desired register. The command is decoded to select the desired register. If the command is a LOAD, the data is gated into the Permuter by means of a special input to the PR Segment Name bus, PRSNB. The PRSNB can be gated to the Pointer Register bus, PRB and then to the Permuter. The desired register is then loaded by selecting the proper output control lines in the Permuter.

Note that although the PR Segment Name Registers are only 16 bits wide, the PRSNB is actually 36 bits wide (4 bytes). This is necessary not only to handle the present situation, but also several other data transfers.

If the command is a READ the TP-Console Interface logic turns on the proper input control lines to the Permuter and then gates the DBB permuter output bus in the control section to the data lines.

1	2	3	4	5	6
---	---	---	---	---	---

# Code

00----	Pointer Register block - one of 15 registers.
001111	Spare Buffer Register.
01----	PR Segment Name Register block - one of 16 registers.
100----	Base Register block - one of 8 registers.
101000	Task Register.
101----	Associative Register block - one of 7 registers.
110----	Operand Stack - one of 8 words.
11100-	Instruction Buffer Register - one of 2 words.
11101-	Name Registers - one of 16 - 4 bit registers.
111100	LR
111101	DR
111110	IR
111111	AR

Figure 4.7.2.11/1 - Register Selection for LOAD and Read Commands

#### 4.7.2.12 Autoload

The AUTOLOAD command causes the TP to initiate its autoload sequence. This involves clearing the entire TP logic and then setting up the various conditions in the TP which are necessary prerequisites for correct operation. The detailed operations involved in this sequence are fully explained in Section 4.8.

Considerable care must be taken in implementing this command on the console. Obviously, we want to make it nearly impossible to autoload the wrong TP. The requirement that a processor be stopped before a console command is accepted helps somewhat in this respect. It might also be desirable to have a manual lockout so that the autoload command is not given accidentally.

#### 4.7.2.13 Position

The POSITION command causes a processor to send to the Engineering Console its current control point position within the control logic. This command can only be obeyed when the processor is halted.

The position of a Taxicrinic Processor is given by two items of information: the currently active sequence and the currently active control point within the sequence which was last activated. The active control point can be easily determined by having the flip-flop portion of each control point or'ed to a particular position in an n line signal array. The particular position will depend on the control points relative position within its sequence. If calling control points are excluded from the signal array, only one signal will be on at a time and the output of the array can be coded into a binary number for transmission to the Engineering Console.

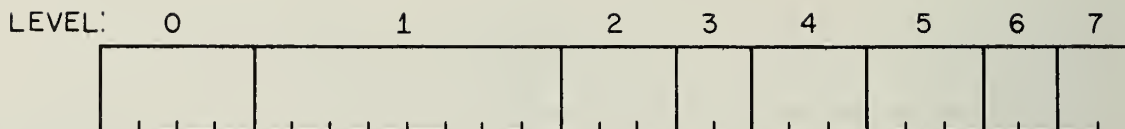
In the actual implementation a 63 line signal array is used. Thus control points may be numbered from 1 to 63. Unfortunately as often happens, errors are found which necessitate adding extra control points in the middle of a sequence. Renumbering all the control points in such a case is quite difficult and may lead to much confusion. As a result the new control points are usually given a number corresponding to a "Nearby" control point but with an alphabetic character, "B" through "D" added to the end. The letter "A" is added to the end of the original control point. In order to handle these letters the signal array provides 3 additional lines for encoding letters. If the control point ends in "B", "C", or "D", the indicator for that particular control point must activate the appropriate letter line as well as its corresponding position in the signal array. If no letter line is activated, the logic assumes the letter "A". These 4 letters are encoded into 2 bits and, when added to the 6 bits from the signal array, produce an 8 bit control point position address which can be sent to the Engineering Console.

The determination of which sequence is the one currently active is more difficult since more than one sequence may be on at one

time and we are really only interested in which one was turned on most recently. The solution to this problem for the TP was to arrange the sequences in a hierarchy based on their calling order as shown in Figure 4.7.2.13/1. Sequences at each level of the hierarchy only call sequences which are at lower levels and only return to sequences at higher levels. In addition the sequences on any given level can only make absolute transfers of control between one another. This means that of all the sequences at any given level, only one can be activated (i.e. entered and not yet finished) at any given time. Thus all of the sequences at a given level can be encoded with one binary number of  $\log_2 N$  bits instead of  $N$  bits (where  $N$  is the number of control sequences at that level).

As a result of this "level recoding", the name of the currently active sequence can be recorded by maintaining an "active" signal for each sequence and then using these signals to produce the proper states in a signal array which consists of several binary fields, one field for each level in the control sequence hierarchy. The currently active sequence will be the one indicated by the highest hierarchical level with a non-zero entry. This can be determined by programming in the Engineering Console once the status of the sequence name control signal array has been sent to it.

Figure 4.7.2.13/2 shows the encoding of the sequence name signal array based on the "Basic Machine" of the TP.



Level 0:	Main Control	0001
	Prim. Inst.	0010
	Imprimitive	0011
	PAU Entry	0100
	PAU End	0101
	PAU Field Acc.	0110
	PAU Fini	0111
	Final Control	1000

Level 1: Instruction Execution Sequences (8 bits)

Level 2:	Name Permutation	001
	Phrase Process	010
	OS Initialize	011
	PAU Access	100
	Post Op. Seq.	101

Level 3:	Increment ICT	01
	Phrase Seq.	10
	IBR Reload	11

Level 4:	Exchange PR-SBR	001
	Stack PR	010
	Unstack PR	011
	OS Read	100
	OS Write	101

Level 5:	OS Clear	001
	OS Entry	010
	OS Clear/OS Entry	011
	SCR Mod	100
	AS Get	101
	AS Restore	110

Level 6:	Memory Add. Calc.	01
	Memory Write	10

Level 7:	Par. Mode Add. Calc.	01
	Queue Counter Update	10
	Memory Read	11

Figure 4.7.2.13/1



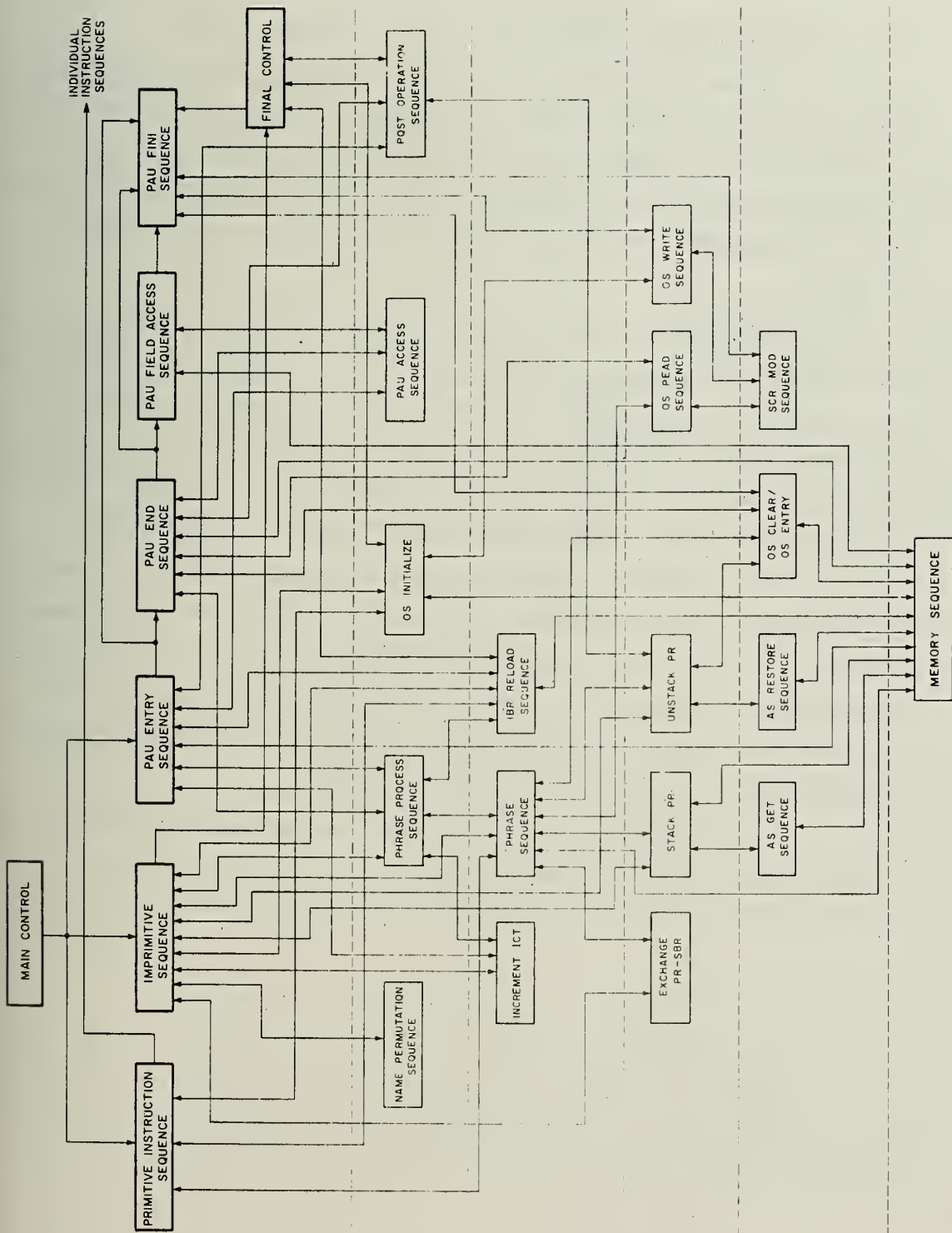


Figure 4.7.2.13/2 Sequence Hierarchy for Basic Machine



#### 4.7.3 TP-Engineering Console Interface - Logical Design

The most important goal for the design of the TP-Engineering Console Interface Logic is that the interface be capable of operating properly even when large sections of the TP are not operating reliably. In order to do this, the interface logic must be reasonably simple and easy to repair and must rely as little as possible on other parts of the TP.

Toward this end, the TP-Engineering Console Interface has been designed almost exclusively with combinational logic. It consists mainly of two sections: the Command Decoder and the Command Execution logic. The Command Decoder is composed only of combinational logic. The Command Execution logic makes use of several control points to set flip-flops and initiate sequences in the case of the Execute Sequence command.

The only part of the TP logic which must be used by the TP-Engineering Console Interface directly is the Permuter. The Permuter and several of the data busses connected to it, must be used in order to get information into and out of the TP registers. Other than this however the interface does not depend on any of the TP logic sections for the interpretation and initiation of its commands.

The remaining portions of this section will describe the details of the logical design of the interface.

#### 4.7.3.1 Engineering Console Command Decoding

The Engineering Console controls the operation of the TP through cables running to the TP-Engineering Console Interface Logic on the individual TP's. A certain number of these lines (at least 10, but possibly more) will be used for the command itself. The purpose of this section is to describe how the TP-Engineering Console Interface logic decodes the information on these lines.

At present the Engineering Console command format can be thought of as consisting of three fields: the processor/unit address field, the command field, and the "operand" field. The processor/unit address field may not be necessary if the parallel structure shown in Figure 4.7.1/1 is used as the Engineering Console communication organization. However, if the series structure is used it will be necessary for the interface logic to compare this field with its own internal name, which might be represented by switch settings which can be changed at the discretion of the maintenance personnel. The logic necessary for this comparison is shown in Figure 4.7.3.1/1.

The command field itself is of indeterminate length at the present time. Given the commands presently mentioned it must be at least 3 bits long. This estimate is based on the assumption that all commands which do not utilize the "operand" field will be given the same command bit configuration and only differ in the setting of the operand bits. This field will be decoded using a full decoder on as many bits as are in the field.

The six bit "operand" field is used for designating sequences in the Execute Sequence and Halt at Sequence commands and is used to indicate registers in the Load and Read Registers commands. In order to minimize the number of IC chips needed for this decoding, the field is first divided into two 3 bit fields which are decoded separately.

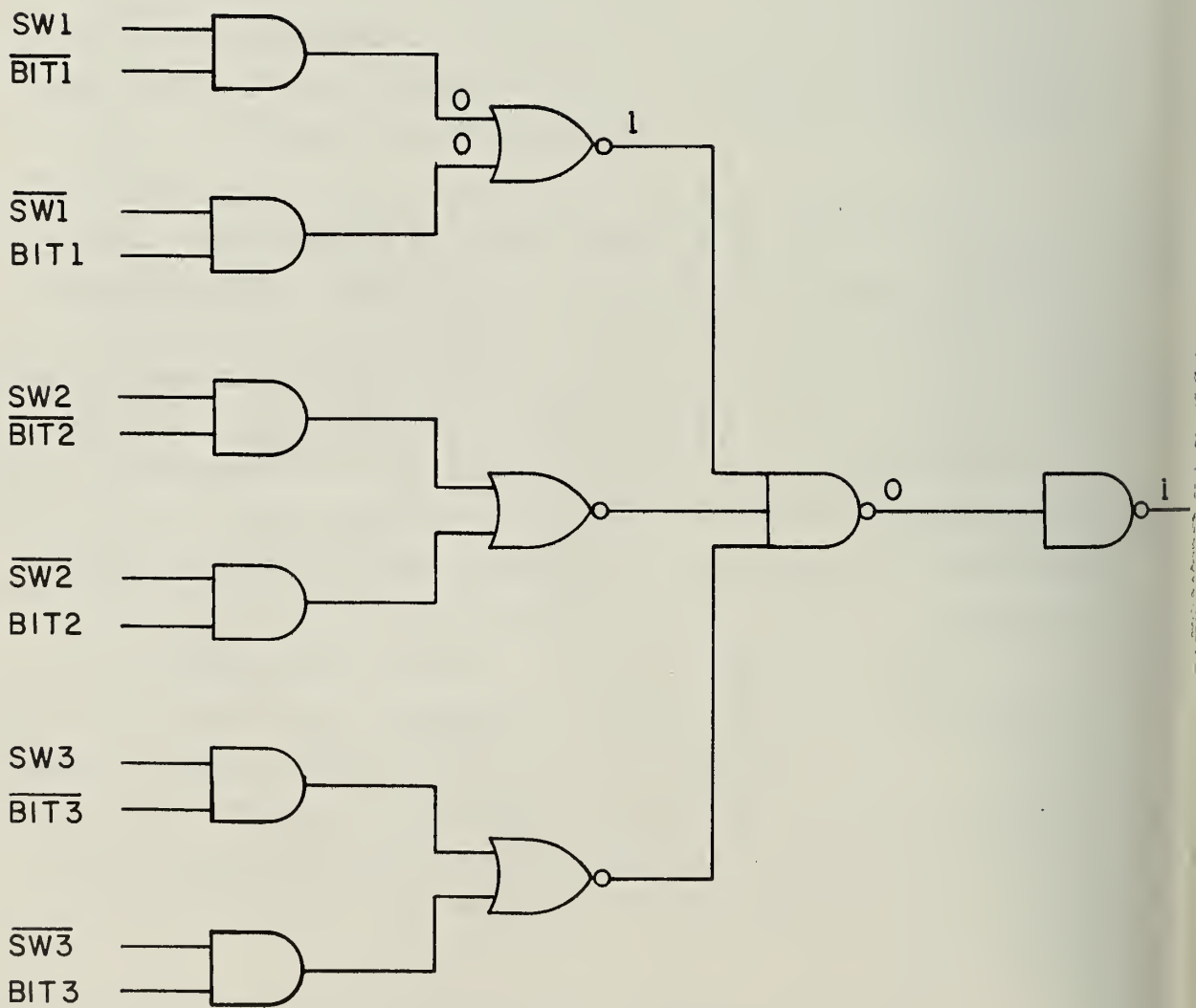


Figure 4.7.3.1/1 - Logic for Detecting Command Address

Then the resulting two sets of 8 signals are combined pairwise in all possible combinations to produce 64 possible outputs each of which represents one of the 64 bit combinations in the 6-bit field. These signals are then used to perform the various necessary selection operations which will be described in Section 4.7.3.2 A schematic representation of the decoder is shown in Figure 4.7.3.1/2.

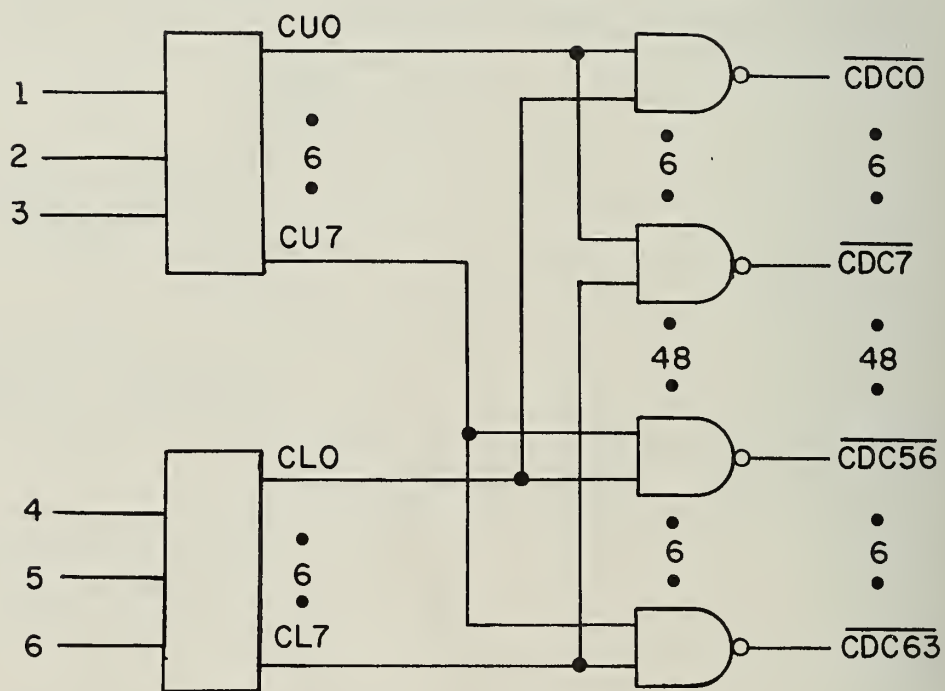


Figure 4.7.3.1/2 - Schematic Representation of the 6-Bit  
"Operand" Field Decoder

#### 4.7.3.2 Engineering Console Command Execution Logic

Once the command from the Engineering Console has been decoded by the TP-Engineering Console Interface, the command execution is reasonably straightforward.

In the case of halt indicators, a two-stage flip-flop is used as shown in figure 4.7.3.2/1. Both stages are set to 1 by the set signal, but only the lower stage is reset by the reset signal. The upper stage is only reset upon the subsequent issuance of a "start" command, such as RUN or Interrupt Return. This technique is used in order to keep the TP from starting off as soon as the indicator is reset. It allows the console computer to pick the type of resumption to be used.

In the case of the Console Interrupt all that need be done is to set the proper indicator signal. The TP Interrupt Sequence does the rest. When the Console Interrupt Return signal is given the interface logic causes the TP Interrupt Return sequence to be activated. Note that both of these commands are exceptions to the general rule that console commands should not directly use sequences in the main TP logic for their execution. This cannot be helped here, however, because of the nature of these specific commands.

The Load/Read Register commands use a fairly substantial amount of combinational logic since there are many possible select signals to activate. If a LOAD or READ Register Command is detected, select signal(s) corresponding to the register indicated in the 6-bit operand field of the command are set. Then the necessary write/read signals are turned on to gate the information into/out of the register. There are many modifications to this general method, however, depending on which register is involved.

For example, in the case of the PR Segment Name Register, two different sets of select signals are needed: one for loading the registers and one for reading. This is not true for the other registers blocks and arises from the slightly different organization utilized by the PR Segment Register storage block.

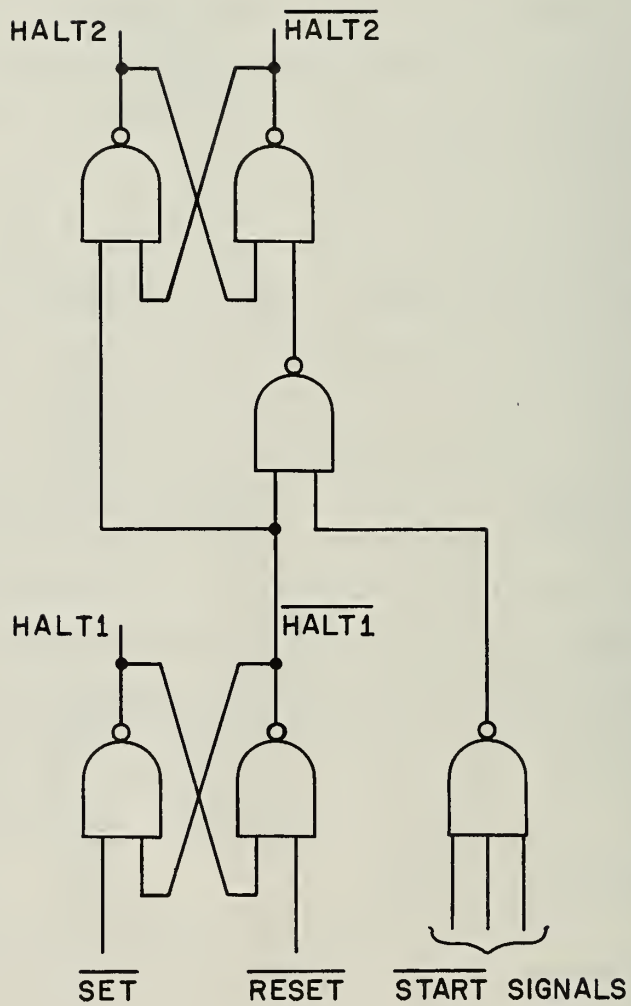


Figure 4.7.3.2/1 Halt Indicator Logic



Base Register selection is also straightforward. In this case the select signals from the console form one of three possible sets of signals which can be used to select the base registers. The other two sets come from the Queue Counters and the Association Logic.

Operand Stack selection is a little different. In this case the operand field indicates which one of the eight possible words in the hardware stack is desired. (Note that these are hardware locations - i.e. not relative to the top of the OS). To implement this it proved necessary to use 8 separate 214-02 circuits to go down to the bottom TP bay. Each of these signals is inverted twice<sup>1</sup> and then dot-or'd with the inputs to the Next Three Byte select logic. These dot-or's are only applied to the  $\overline{\text{BY0}}$ ,  $\overline{\text{BY4}}$ , ...  $\overline{\text{BY28}}$  signals since using these, the proper 4 bytes will automatically be selected in each case.

It should be noted that since the words in the OS are being chosen on "word boundaries", the special TP logic which inhibits OS writes past the OSTR position will not have any effect (see Section 2.3.1.3).

The Pointer Register selection logic utilizes the same type of dot-oring technique except that its selection signals are dot-or'd with the select signal outputs from the Name Register - Name Bus Compare logic.

In the case of the Instruction Buffer Register selection the gating of the ICT to the IBR selection decoder must be inhibited by activating  $\overline{\text{IBRC/S}}$ . Then either the constant zero or four can be placed on the input to the decoder to fetch either the left or right half of the IBR, respectively.

The Name Registers present a particular problem. They can be read out easily enough by using the logic which has been provided for use

- 
1. The double inversion is necessary because 214-02 circuits may not be dot-or'd. This implementation, however, is still faster than if some other circuit had been used to drive the cable to the bottom bay and had been dot-or'd directly.

during interrupts. Unfortunately there is no direct way to load these registers. The only way is to indirectly load them by first loading the Shadow Name Registers (SNR's) and then gating the SNR's to the Name Registers. This would destroy the values in the SNR's however, and if the TP were in the middle of an imprimitive sequence name permutation this could be fatal.

At the same time that the proper select signals are being activated, the control signals for reading or loading the registers are also turned on. In most cases this merely involves setting the proper read/write signals for the storage blocks or registers desired. In the case of a load operation it is also necessary to gate the data lines to the PRSNB bus and to gate this bus to the Permuter input.

#### 4.8 Turn-On and Initialization

Turn-on and initialization consist of the processes involved in getting a TP without power into a state in which a task can be run on it. The purpose of this section is to describe, first in overall terms and then in detailed logic, exactly what must be done to accomplish this transformation.

The turn-on and initialization process can be broken down into four distinct phases: Power Turn-On, Register Clearing, TP Hardware Initialization, and TP System Initialization. Power Turn On consists of those operations which must be performed in order to successfully apply power to a Taxicrinic Processor. Register Clearing involves clearing all the TP registers as well as all the various flip-flops and indicators. TP Hardware Initialization is the process whereby registers are loaded with the information necessary for a TP to be able to run. TP System Initialization involves loading several system constants so that the TP can begin to execute the proper supervisor procedures to allow it to be assigned a task.

#### 4.8.1 Power Turn-On

Power turn-on consists of those operations which must be performed to successfully apply power to a Taxicrinic Processor and to set up the computer hardware for the TP Register Clearing stage. This latter process consists mainly of ensuring that all control points are in the cleared state.

The major problem of power turn-on from the logical design point of view is to prevent the various random initial states of the machine when it is first turned on from initiating unintended operations in the logic. In order to do this the GOTASK and GODELY signals should both be forced to zero as soon as possible after the power is turned on. This will prevent sequences of unintended operations from propagating through the control logic. It will also prevent the task signals of all control points from turning on. Thus no control signals will be activated except for transient pulses directly due to the power turn-on itself.

Once the power has settled down the GOTASK and GODELY signals may be turned on and a negative pulse can then be sent to the TP Hardware Initialization sequence to begin its operation.

#### 4.8.2 Register Clearing

The Register Clearing phase of the TP Turn-on and Initialization process consists of ensuring that all control points are in the cleared state and that all registers and flip-flops are set to zero. The main operations consist of activating the common clear signal for all of the control points, generating a zero on the Distribution bus and loading all the registers with it, and clearing the flip-flops.

In the case of clearing groups of registers such as the PR storage block it may be necessary to have a counter which is incremented and used to select each register in turn to be loaded with zero.

### 4.8.3 TP Hardware Initialization

The TP Hardware Initialization process is performed after all the control points have been reset. One of its primary purposes is to load the RP Name Registers with unique names ranging from 0 to 14. Another is to initialize the associative registers queue counters. In addition, certain other operations are performed.

The PR Name Register initialization is performed by an initialization control sequence which makes use of the ACT and CCT counters, the Permuter and the IR arranged in the order shown in Figure 4.8.3/1. The ACT and CCT are initially set to zero and one respectively and then gated to the positive constant generating signals of the Permuter. This causes the low order byte output of the Permuter to contain the contents of the two counters. This output is then permuted and merged into the IR under control of the middle two bits in the 4 bit ACT.

After one byte has been loaded, both counters are incremented by two and the process is repeated, this time loading the constants into a new byte. After 4 cycles, the IR will be loaded with 8 numbers, packed two to a byte, which is the same format used for storing the name registers during an interrupt. The interrupt return circuitry is then used to load these numbers into the first 8 name registers.

The final 7 name registers are loaded by repeating the same process while the counters are counted the rest of the way to 16.



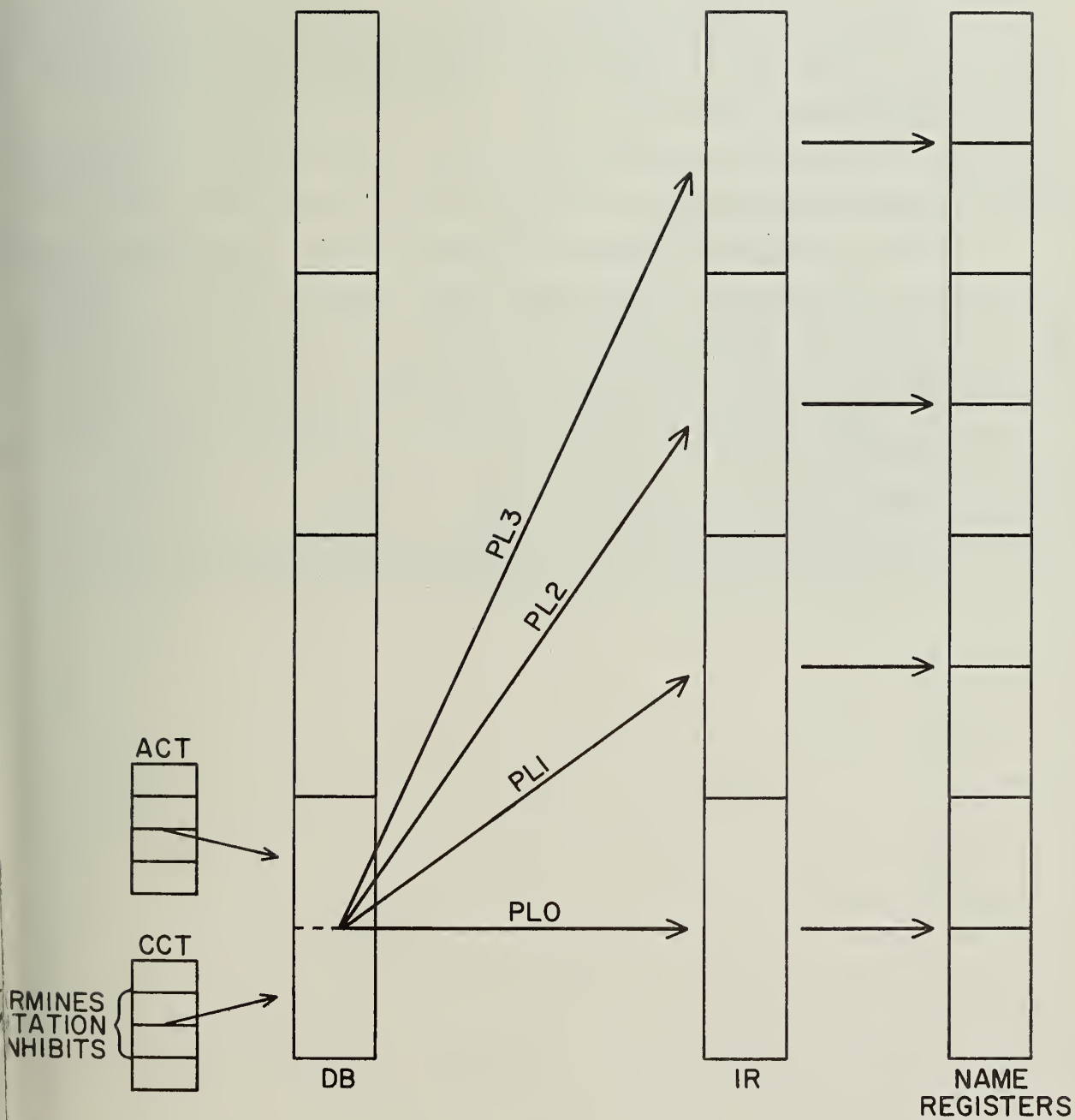


Figure 4.8.3/1 - Name Register Initialization



#### 4.8.4 TP System Initialization

Once the TP Hardware Initialization phase is completed, the TP must be loaded with enough information so that it can become activated and successfully join the system. This is the job of the TP System Initialization phase.

The main operations which must be performed are loading BR#0 with the base descriptor of the Supervisor task and then issuing the TP and Activate Interrupt (see Section 5.6.5.2). Both of these operations are performed by the Engineering Console, and therefore the details will not be described here. It is sufficient to say that the Engineering Console is programmed so that it can start up a TP once it reaches the TP System Initialization phase simply by issuing the proper load commands and then giving the Exchange Net an ACTIVATE command with the proper initial address to cause the TP to begin the supervisor reactivation sequence. (This could also possibly be done by having the Engineering Console start up the TP Activate sequence directly).

#### 4.A Control Point Logical Design

Beginning with Section 4 of this manual, the largest part of the logical design consists of sequences of operations. Each control sequence in the Taxicrinic Processor has been broken down into control steps. These steps consist of task logic which performs the necessary operations by activating the proper control lines, and decision logic which determines which control step will be activated next. The control steps are performed by logical circuits called control points. Generally speaking each control step is performed by one control point.

The purpose of this section is to explain the design and historical development of the control points used in the TP and to describe some of the more standard techniques which are used in the implementation of control sequences. In the following subsection, Section 4.A.1, the historical development and final design of the elementary control point circuit is described. Section 4.A.2 describes the design and use of the special calling control point while Section 4.A.3 goes into the various design techniques used in implementing control sequences.

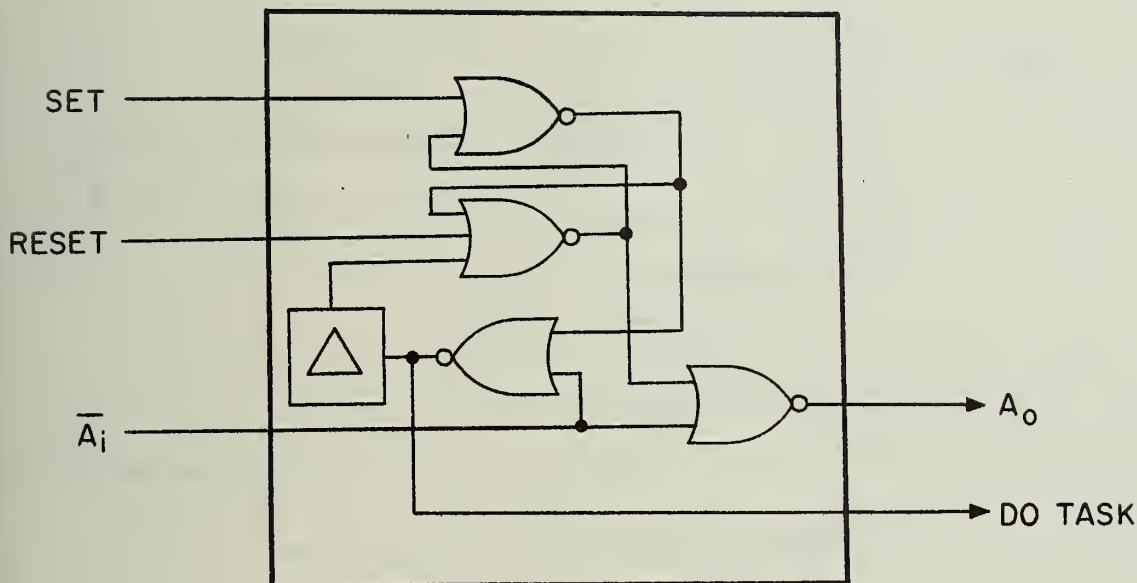
#### 4.A.1 Design of the Control Point

The control point design for Illiac III has had a long and tortuous history. It originally developed as a modification of the Illiac II "speed independent" control circuits.<sup>1,2,3</sup> The basic idea behind a control point is that it initiates some operation by turning on a given set of control lines and leaves these control lines on until it either receives a reply indicating that the operation is over, or until a certain length of time passes.

In general, a control point begins operation when it receives an advance in signal,  $A_i$ . It then performs the operation it is designated to perform by activating a task signal,  $T$ , which in turn activates the necessary control lines. Finally, when the operation has been completed, the advance out signal,  $A_o$ , is activated which in turn will activate the next control point.

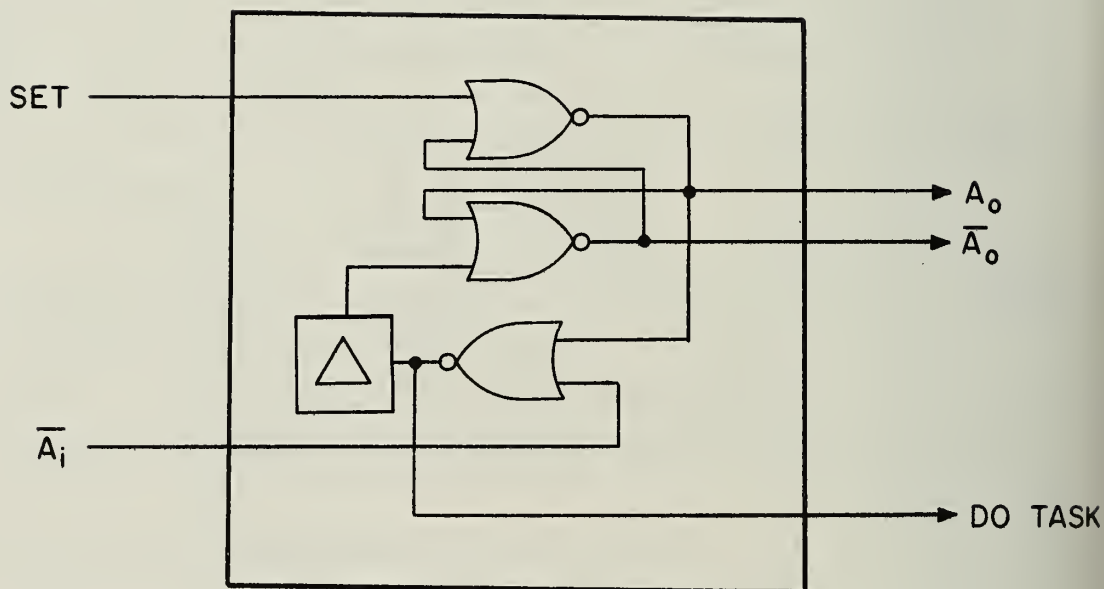
All of the control point designs which have been considered for the Illiac III system are centered about a flip-flop which determines the state of the control point. However, the way in which this flip-flop is used has been subject to at least two different philosophical interpretations. In the first interpretation the signal returning from the completed operation (or from the time delay model of the operation) is directly used to reset the flip-flop which in turn directly drives the advance out signal. The advance out signal then remains on until the control point is reset. Examples of this type of use are shown in Figures 4.5.1/1 through 4.5.1/4. (text continued on page 6)

1. Gilles, D.B., "A Flowchart Notation for the Description of a Speed Independent Control", Department of Computer Science File No. 386, August 1961.
2. Robertson, J.E., "Problems in the Physical Realization of Speed-Independent Control", Department of Computer Science File No. 387, August 1961.
3. Swartwout, R.E., "One Method For Designing Speed-Independent Logic for a Control", Department of Computer Science File No. 388, August 1961.



This circuit, originally designed by K.C. Smith, contains a reset signal as well as a set signal. This allows the designer to develop logic sequences which skip certain operations simply by resetting the desired control points before entering the sequence. A possibly undesirable feature of this circuit is that in order to maintain the outgoing advance signal, the incoming one must remain active. This means that, unless additional flip-flops are provided outside of the normal control point logic, the designer will run into a problem whenever any part of the sequence is turned off since this turnoff will propagate forward and eventually stop the operation of the control points.

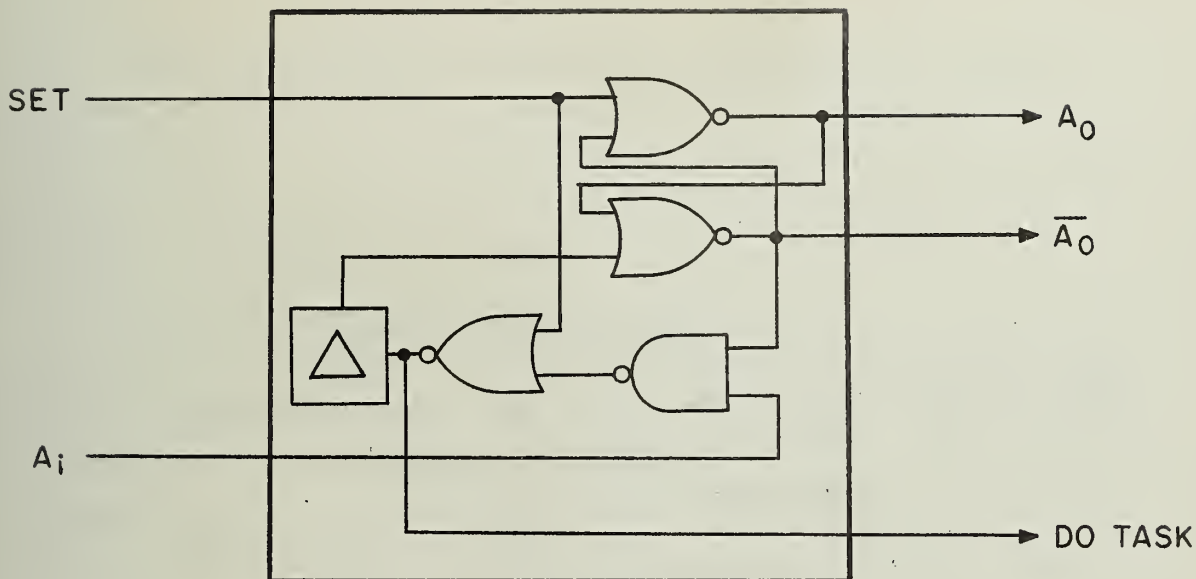
Figure 4.A.1/1



The above circuit was an attempt to control the "broken chain" effect mentioned in Figure 4.A.1/1. In this circuit the control point flip-flop is directly used to generate the advance signal. Thus once the control point task is completed the control point will continue to produce the advance signal until the control point is set for the next incoming signal. Note, however, that we have lost the ability to skip around control points by using a reset signal. This cannot be allowed in this circuit since a reset immediately triggers the advance signal to the next stage even independent of any input. Note also that in this circuit the number of logic elements has been reduced to 3, 2 input NOR's and that both true and complement advance signals are available.

Figure 4.A.1/2

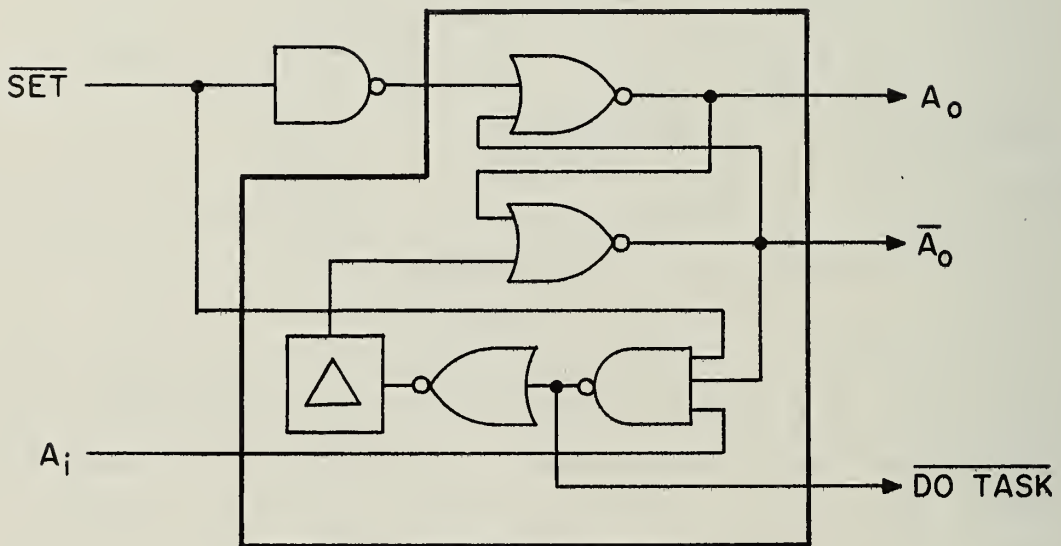




One problem with both of the circuits shown in Figures 4.A.1/1 and 4.A.1/2 is a race condition which develops if several control points in a sequence are reset simultaneously. In this case the incoming advance ( $\bar{A}_i$ ) may not go to "1" fast enough to prevent the task from starting when the control point flip-flop is set causing the other input to the task driver to go to zero. This means that in order to reset a series of control points without creating spurious task signals it would be necessary to set a given control point only after its incoming advance signal had gone to its inactive state. This would cause a considerable timing problem in many cases.

In order to solve this an extra NAND can be added as shown above. Now the set signal is also used to disable the task signal while the flip-flop in the control point is being set. All spurious input advance signal fluctuations are ignored. Thus in order to set a series of control points it is only necessary to generate one set signal for all of them and then keep it on long enough for the transients in the logic between the control points to die out. Note that the input advance signal is now a positive signal instead of a complement and it takes an extra collector delay between the time the input advance activates and the do task signal starts.

Figure 4.A.1/3



One problem with the circuit in Figure 4.A.1/3 is that the loading caused by the task signal will vary from one case to the next and thus the time to charge the capacitor will depend on the load. In order to solve this problem the above control point was designed. In this case there is only one load on the capacitor no matter what the task load is.

Figure 4A.1/4



The main ~~problem~~ with this type of control point is that it must be specifically set to an accepting state before being used again. The main advantage is that the advance out will remain on until it is reset. This allows the  $A_0$  signals to be used for a variety of purposes where a signal must be maintained for an indeterminate amount of time.

The second type of control point implementation, which is shown in Figure 4.A.1/5, is organized so that each control point automatically sets up the following control point. In this implementation the advance in signal transitions become important. Just prior to using a control point the advance in signal is activated to set the control point flip-flop for operation. However, the task signal does not become active until after the advance in signal returns to its unactivated state. This means that the reply signal in the previous stage can be used to set up the stage coming after it. Note that in Figure 4.A.1/5 the reply goes to zero before stage B becomes active.

In addition to automatic setting, this control point implementation has the added advantage of allowing a greater margin of safety between the time the task signal turns off and the time the advance out signal is activated. In the versions shown in Figures 4.A.1/2 through 4.A.1/4 the advance out signal was actually turned on before the task signal was turned off. This might accidentally cause two operations to be on simultaneously.

The main disadvantage is that the advance out signal does not remain active for a very long time (only as long as it takes the task signal turn-off to propagate through the model or the return signal generator). This means that the advance out signal cannot be used as a long term data line for the control of the operation sequencing, e.g., it cannot be used to determine a branch at some point further on in the sequence. Thus all decisions involving the advance out signal must utilize decision signals which have settled down at the time the advance out is activated and must activate logic which will accept pulse inputs.

Eventually this last control point implementation was chosen for use in the TP, mainly due to the advantage of automatic resetting. This also ensured that the TP and AU would use the same type of control point.

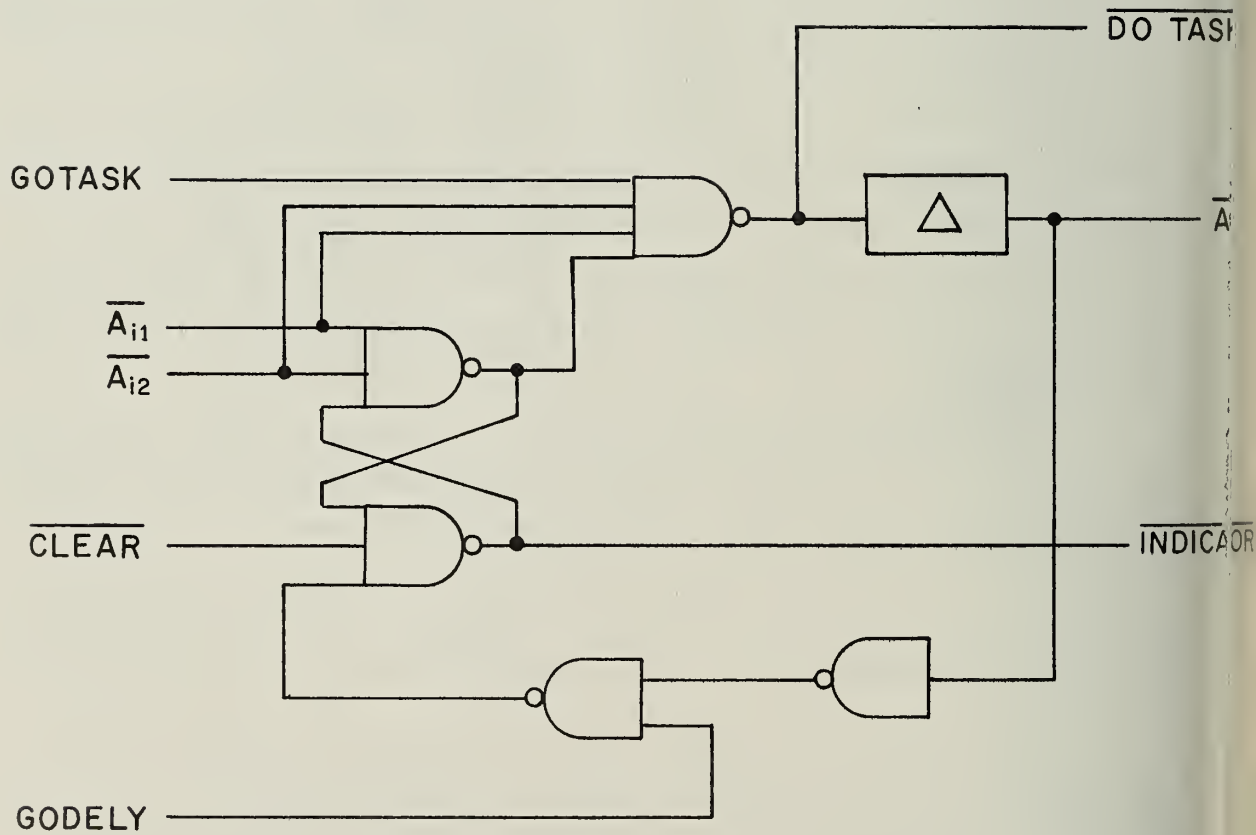


Figure 4.A.1/5 Final Control Point Design

#### 4.A.2 The Use of Standard Control Points

The standard control point as described in the previous section, is the principal sequence controlling element used in the control sequence logic. For ease of use, it is represented on the main logic drawings by one of the symbols shown in Figure 4.A.2/1. Note that the various input and output signals are indicated by letters at the points they enter and leave the box. The CLEAR and GOTASK signals are omitted from the signals since their use is always the same and their presence would have cluttered up the symbol. They are always connected to their respective input pins on the card, which have been standardized as pins Y and 20, respectively, on all TP control logic drawings. The upper part of the symbol is divided into either two or three boxes (depending on the control point variant being used) which represent the chips used to implement the control point. The location of these chips on the circuit board is indicated by the letter and number at the center of each box.

The four boxes in the lower part of the symbol are used to describe parameters for the control point. The leftmost box contains the control point delay in nanoseconds. The second box will contain the capacitor value needed to generate this size time delay. The third and fourth boxes contain the control point variant name and the unique name for that particular control point, respectively. The variant types are described by a 3 or 4 character code in which the first letter represents the general type of control point, the second and third numbers represent the variant number and the last letter (always a D, if it is present at all) indicates that a speedup diode is used.

In general the individual control point names are classified into functional groups corresponding to the functional grouping of the sequences. Each group is assigned one or two characteristic letters (i.e. M for memory sequence, MU for Main Utility sequences). These letters are prefixed to all of the signals related to each individual control point. For example, the main utility sequences use control points whose names are designated MUn where n is a 1 or 2 digit number optionally followed by a single letter a, b, c, or d. Thus, if it is necessary to assign a name to any of the signals directly attached to one of those control points, they would conform to the following format:

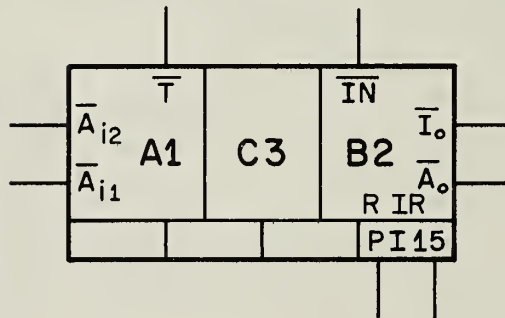
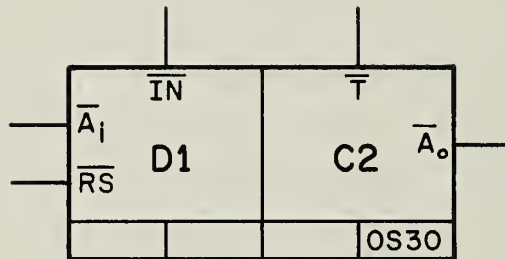


Figure 4.A.2/1 Control Point Symbols

Advance In	- $A_i$ - MUAIn
Advance Out	- $A_o$ - MUAOTn
Indicator	- IN - MUNn
Interrupt Out	- $I_o$ - MUIOTn
Task Signal	- T - MUTn

The control logic drawings themselves are arranged on a functional basis with each series of related sequences being assigned to a particular set of drawings beginning with drawing 30. The 30 to 39 groups are assigned to control logic involving the basic machine sequences while the drawings beginning with 40 and above are assigned to the control logic for the instruction sequences.

Each control logic card is represented by two drawing sheets, a control logic drawing sheet and a control point drawing sheet. The control point sheet contains the detailed logic and wiring for each control point on the control logic sheet. Its purpose is simply to give full representation of the control points and need not necessarily be included in every drawing collection.



#### 4.A.3 The Use of Calling Control Points

Almost all of the operations of the TP control are in reality subsequences of operations which are "called" by some other sequence. The obvious reason for this type of structure is that it reduces the logic by allowing common operations to be done using the same physical logic. Another advantage is that the control logic is easier to visualize conceptually when it is placed in such a modular format, and is also easier to debug and check out.

The general philosophy used in the sub-control logic sequences is quite analogous to subroutines in software. Each sequence may have certain control flip-flops and/or control signals which are set to their desired states sometime before the sequence is activated. Activation of the sequence is caused by the use of a modified control point with a genuine reply signal, such as the one in Figure 4.A.3/1.

The actual operation of the simplified calling control point is fairly straightforward. When one of the advance-in signals goes to "0", the flip-flop is set. Then when that signal returns to "1", the task signal turns on (i.e. goes to "0"). This signal is used to activate the advance-in signal of the first control point in the subsequence which is being called. From this point on, the subsequence has control. Note that in addition to starting the sequence, the calling control point task signal can be used to set control signals which will remain on for the full duration of the sequence.

It should be noted that since the task signal only makes one transition before the return signal arrives, it is necessary to control the advance-in signal to the first subsequence control point so that when it moves from '0' to '1' the task signal is activated. This means that the first control point in a sequence will normally rest in the set state. Thus it is extremely important to avoid having any decision logic on the first control step of a sequence which might depend on signals which could change during the sequence. If this were the case, a change in one of these signals would cause the input of the first control point in one of the alternate paths to go from '0' to '1' and cause the sequence to restart without a task signal from a calling control point turning it on.

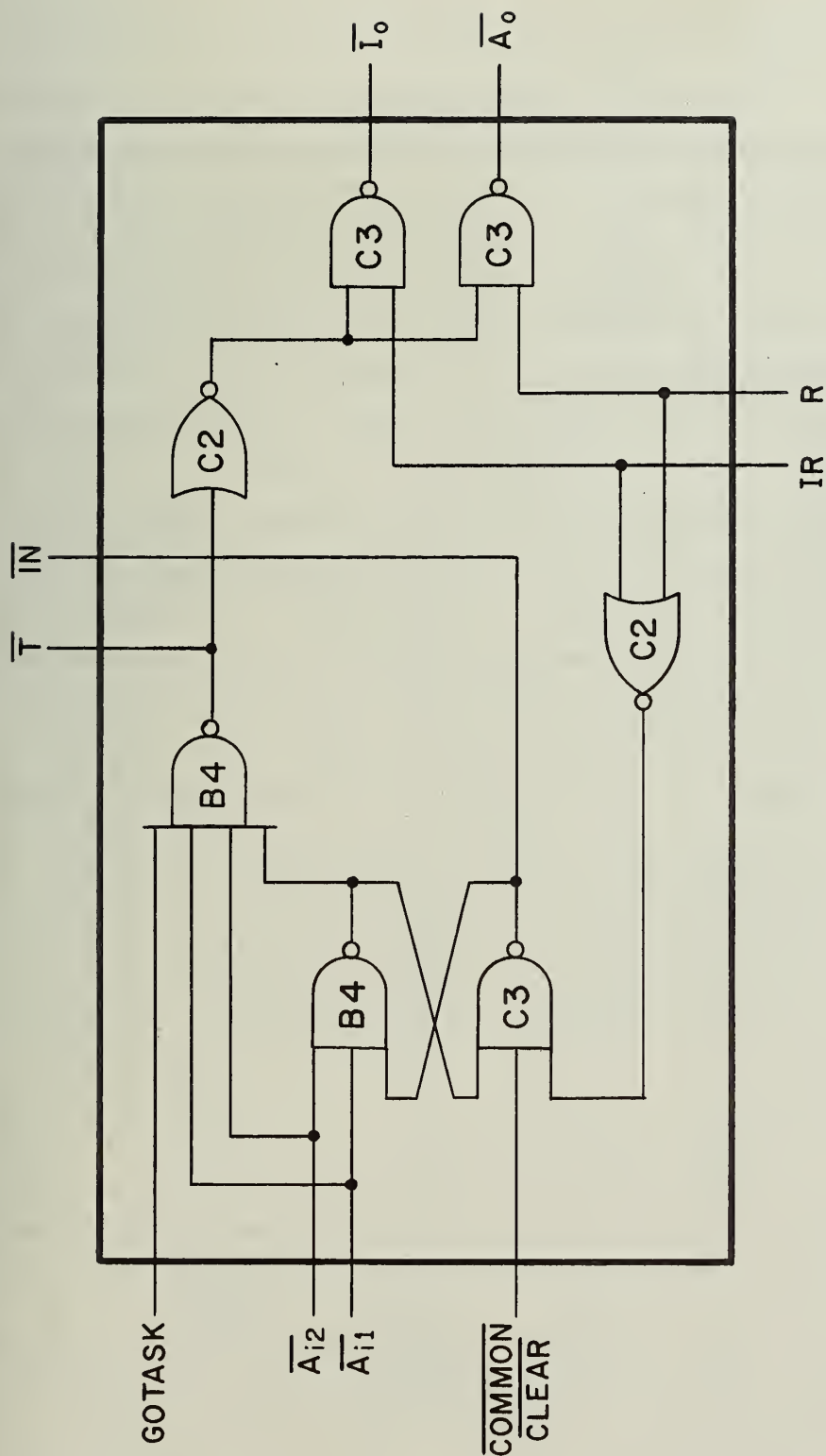


Figure 4.A.3/1 Simplified Calling Control Point



Once the sequence has been started it will continue to run until it reaches a point where it must return. In general there can be either of two types of returns: a normal return, from which the calling sequence proceeds in a normal fashion by activating the advance out signal,  $A_O$ , and the interrupt return, from which the calling sequence may have to perform some special repair sequence which is started when the Interrupt Out signal,  $I_O$ , is activated. The type of return which has been performed is differentiated by the signal which the returning subsequence activates: R for a normal return and IR for an interrupt return. Note that either of these signals will reset both control point flip-flops, but that each one only activates its own corresponding "out" signal.

Note that there are no GOTASK or GODELY signals. It was decided to exclude this feature in calling control points because of an additional problem which they would cause. In particular, if the GODELY signal is off when one of the return signals activates, the signal will be lost because they are essentially pulse signals (i.e. from the  $\overline{A_O}$  of the last control point in the sequence). When the GODELY signal goes to 1, the return is no longer active and therefore the control point is stopped for good.

There are several possible solutions to this problem. One involves adding flip-flops which would "save" the return pulses. Then if GODELY is off when they return, the flip-flops would stay on until GODELY comes back on. It was felt however that the additional logic was too much trouble to go to just to allow a halting facility, so the solution was not adopted.

The calling control point shown in Figure 4.A.3/1 is the type most commonly used in the TP. However there is another type, which is shown in Figure 4.A.3/2, which is used in the case of those sequences which may be active when the interrupt operations are performed by the TP.

In order to understand the philosophy behind this more complicated type of calling control point it is perhaps best to re-examine momentarily an ordinary control point. An ordinary control point has associated with it one flip-flop which may be in either of two states: reset, in which the control point remains off, and set in which the control point is either preparing for or activating a task signal. In certain calling control points, however, there may be more than two states. These are the

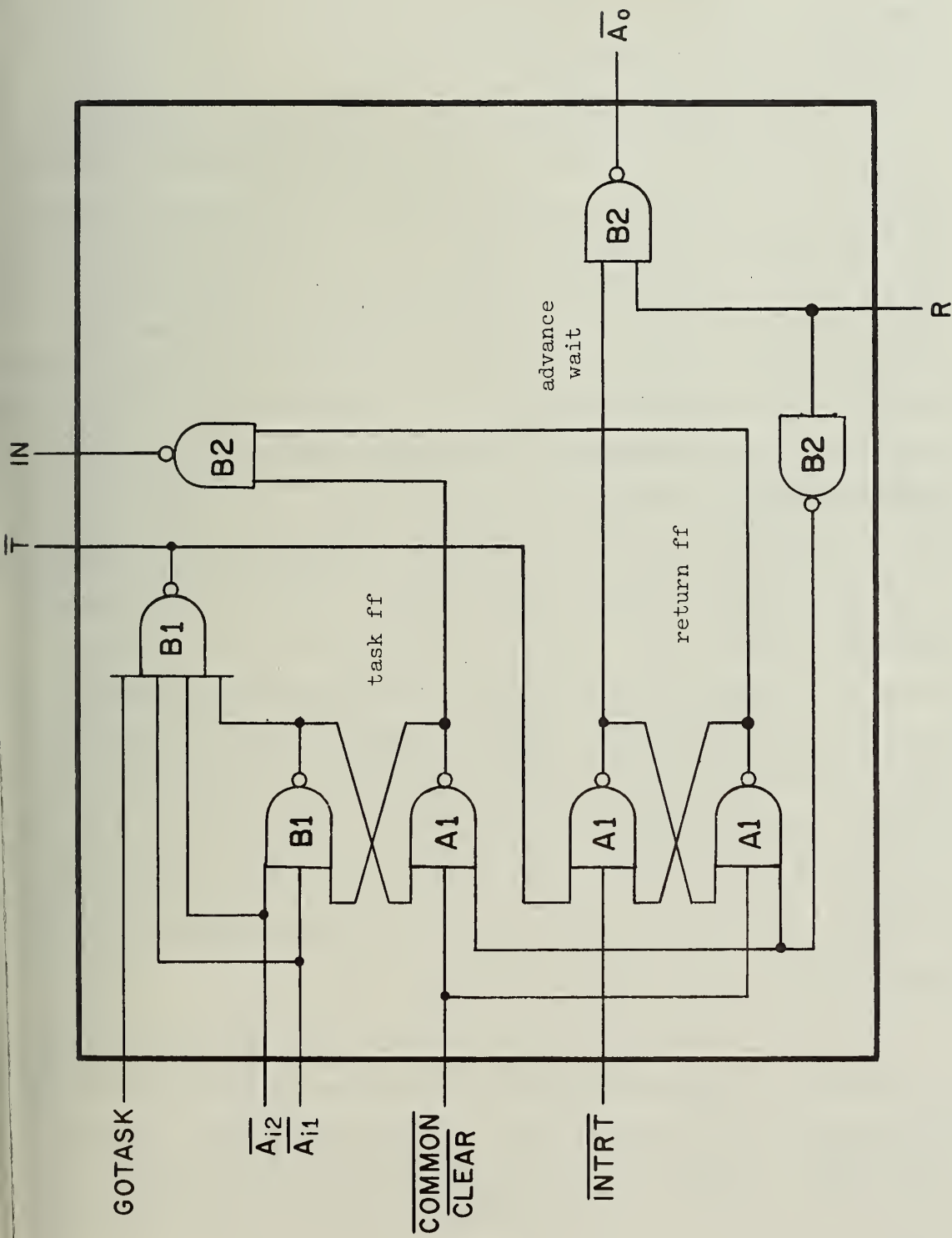


Figure 4.A.3/2 Status Saving Calling Control Point

reset state in which it is inactive, the normal active state in which the task signal is turned on and the interrupt return active state which is produced when the TP returns from an interrupt.

This latter state needs a fuller explanation. As discussed in Section 4.6, the basic idea used when a TP has detected an interrupt condition is to try to undo as many unfinished operations as possible. This means that if an interrupt occurs in a subsequence, it will try to restore the TP to its state before that subsequence was entered. If the subsequence can do this, it will make an interrupt return, and the sequence which called it will have to try to do the same thing. Eventually however, it will become impossible for a given sequence to do this because of some permanent change in some part of the data. When one of these interrupt points is reached, a call is made to the interrupt sequence using a status saving calling control point. The status of all such calling control points is saved during an interrupt so that after the interrupt has been processed, the logic will know which control point called the interrupt sequence.

In addition to the location of the interrupt return point, however, it is also necessary to know the series of calls by which control came to be transferred to that point since after the subsequence is restarted it must know where to return when it is finished. This is done by also storing states of those control points which might be active when an interrupt return point is reached. All of this storing is done by gating out the control point status by means of the CP Save signal into interrupt storage memory.

Once the interrupt has been processed by the appropriate program an interrupt return instruction will be executed to restore the TP hardware to its status at the time it reached the interrupt point. In order to do this the calling control points which were on at that time must be set once again. However this must be done in a manner which will not initiate the respective sequences.

Thus, the meaning of the third calling control point state mentioned above becomes clear. In this state the task signal is off, but the control point remains in a wait state for the return signal. Note that

since all interrupts will be handled by the called sequence, there is no interrupt return signal.

With this in mind we can now refer to the logic drawing of a calling control point as shown in Figure 4.A.3/2. Note that there are two flip-flops both of which are set by either advance in flip-flop and both of which are reset by the clear and return signals. The four possible states are assigned as follows:

state	task ff	return ff
reset	0	0
set-waiting for activate	1	0
normal active	1	1
return active	0	1

The ordinary operation of the status saving calling control point is the same as in a simple calling control point in the sense that if there is no interrupt, the return signal will eventually reset both flip-flops and activate the advance out signal. If there is an interrupt, the Interrupt Sequence will save the status of the calling control point before giving control to the Interrupt Handler Procedure. When the interrupt has been processed it will eventually give control to the hardware Interrupt Return Sequence which will restore the status of the status saving calling control points by clearing all control points and then activating the proper individual INTSET lines to each control point on the basis of the information previously saved in the interrupt block. This causes the return flip-flops of the appropriate control points to be set. Then control is returned to the instruction control sequence by activating the return line for the Interrupt Sequence. This line goes to every calling control point which might have called the Interrupt Sequence, but since only the one which made the original call will have its return flip-flop activated, it is the only one which will activate its advance out signal and thus the interrupted sequence will continue. In the same manner control will continue to be passed back at the completion of each sequence by means of the return signals and the activated return flip-flop.

It should be noted in passing that after processing an interrupt in the above manner and returning it may be possible to have a second

interrupt and in these cases the status must again be saved. Thus it is necessary to have the indicator signal on if only the return flip-flop is on. Since it is also necessary, for diagnostic purposes, to have the indicator on in the case where the GOTASK has been turned off, it is necessary to have the indicator signal equal to the OR function of the task and return flip-flops.



#### 4.A.4 The Design of Control Sequences Using Control Points

The purpose of this section is to describe the techniques which were developed during the beginning stages of the TP Control Sequence Logic Design. In this section we will discuss some general techniques which can be used regardless of the specific type of control point being used. Thus no reference to specific implementations will be discussed.

Given a flow chart in terms of basic operations of the sequence to be implemented, the design of the control point logic to implement that sequence is fairly straight-forward. If we assume that the flow-chart operations consist of those operations which can be performed by activating various low level control signals such as gates, enables, inhibits, etc., then the design can be expressed as a step-by-step process.

The first step is to redraw the flow chart in a very explicit form (called the "control step" flow chart) so that every control signal which must be influenced is expressed. Thus if a particular operation consists of gating from one register to another while inhibiting certain bits, all the necessary gating signals which must be activated to accomplish the operation should be written down. Each set of basic operations which can be performed in parallel should be written as the contents of one operational block or control step of the flow chart. These control steps thus become in effect the list of control signals which must be turned on by the task signal of a given control point. The decision logic between control points will be represented by the various decisions in the flow chart.

After the control step flow chart has been drawn as described above, careful inspection will reveal that many single control steps or even whole sequences of control steps are identical. These are often capable of being implemented by the same physical control point or points. Such occurrences should be noted for future reference.

The next step is to draw out the actual control logic. This basically involves converting every control step represented in the flow chart to a control point, producing the necessary decision logic between control points, and designing any additional combinational logic which might be necessary for the operation of the sequence. In addition, the

designer must take care to satisfy the pin and chip limitations of the cards. Generally, we have tried to keep each sequence on one card or some whole multiple of cards. This allows easier debugging. Occasionally, however, it has been necessary to have parts of two different sequences on the same card.

The task signals from the various control points on a given drawing (i.e. card) are usually connected directly to output pins. If conditional task logic is necessary, it is included on the sequence card itself and the additional task signals are also connected to output pins. The task signal outputs are labelled simply as task signals if they are unconditional and are followed by a letter at the end of the alphabet (u through z) if they are conditional or are "or" combinations of other task signals. Note that these names should be made to correspond to the name labels for the various boxes on the control step flow chart.

Several redrawings of the control point logic will probably be necessary in order to optimize the logic between control points and to determine the optimal number of control points.

Once the sequence control logic appears to be complete and reasonably distributed over as many cards as is necessary, the design of the task drivers can be started. The task and conditional task signals from all of the cards included in a particular set of logic drawings (i.e. with the same initial 2-number name) are collected on 3 to 5 task driver cards, which are included at the end of the logic drawing set. These cards collect all the various task signals from the set which turn on each particular control line and produce one output for each control line activated by the set. These outputs are then connected with the outputs of other drawing sets (usually by means of dot-or's) to produce a single control line going to the proper logic to be activated. Thus we have a modular means of handling the very large fan in of control task signals to the control lines themselves.

Figure 4.A.4/1 gives a simple example of a task driver card. Note that it consists of three levels. The first level gathers together barred task signals while the second level gathers the output of the first level and any other unbarred task signals by means of dot-or connections. The final stage drives the actual control line. This will usually be a



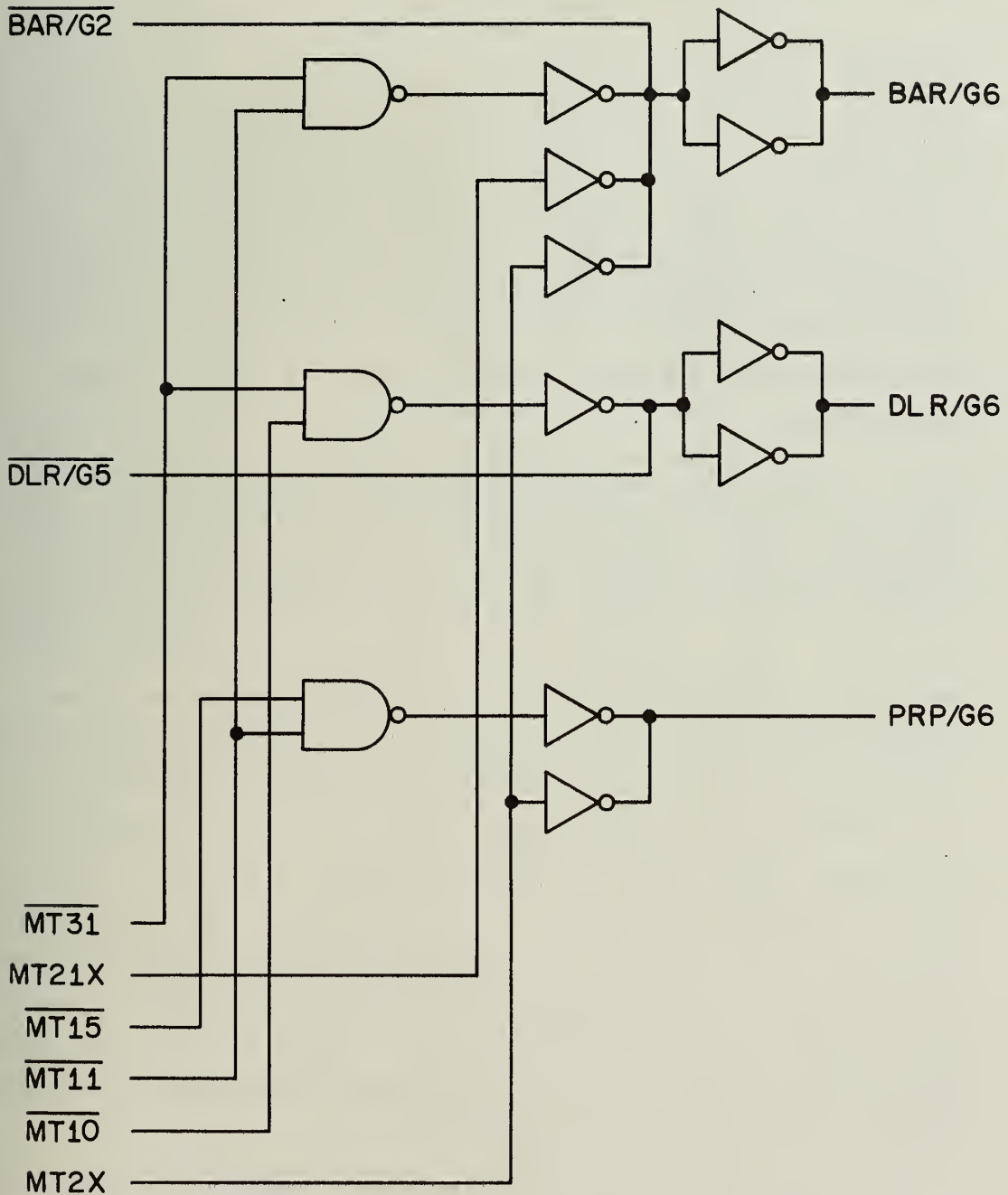


Figure 4.A.4/1 Example of Task Driver Logic

"cable driven" line, i.e. it will have cable terminations at both ends.

Note that the final stage is omitted for some signals. In these cases the output line will be going to some other set of task drivers and will be dot-ored at the second level in that set. The basic idea is to try to maintain an equal delay in all of the task driver paths. In Figure 4.A.4/1, these signals are represented as BAR/G2 and DLR/G5. The names on the task driver cards follow a simple convention, namely in all control signals with a slash and a final letter, a number indicating the drawing set is added at the end of the signal. If the control line does not have a /G, /N, or whatever, /Ei is arbitrarily added to the end where i is this same set number.

The hardest part in designing the task signals is partitioning the output signals on the cards since in order to minimize the total number of pins used on all the task driver cards, it is necessary to place the drivers for control signals which are often turned on by the same control point task signals together on the same card. This minimizes the number of cases where a single task signal from a control point has to get to many different task driver cards. Unfortunately the partitioning job can be extremely time consuming. As a result the general tendency has been to do a quick and dirty partitioning based on an a priori knowledge of which of the most common control signals get turned on together and then putting the less commonly used signals in whatever spaces seem most convenient. This is the result of the realization that the cost inherent in, say, using one more card than necessary in a non-minimal design will be a lot less than the cost in time of trying to get a minimum partitioning, especially since the task driver logic has a very high probability of being changed due to additions or deletions in the logic.

The general partitioning method is based on the use of a table which lists all of the signals (in alphabetical order) along one border and all the task and conditional task signals (in numerical order) along the other border. Check marks are then made along each column to indicate if a given task signal must turn on a given control signal. From this table it is possible (with varying amounts of effort) to decide how to group the control signals so that each task signal will have to go to a minimum number of task driver cards.

The final step in the control sequence design process is to update the original control step flow chart so that it reflects the final logical design. This is helpful as a final check and it also produces a document which can be used as a reference in checking out the logical design at a later stage. In particular each box in the control step flow chart should be labelled with the name of the task signal or conditional task signal which it represents. If it represents more than one signal, the respective names should be grouped together and separated from the other groups by dotted lines and each group should be separately labelled.

In its general layout, the control step flow chart should follow the logical design, i.e. the parts of the flow chart corresponding to each logic card should be delimited by dotted lines across the entire flow chart. Lines crossing these delimited areas as well as those coming from entry and exit circles represent specific control lines on the logic cards and should be so labelled. This allows for easy comparison between flow chart and logic.



U.S. ATOMIC ENERGY COMMISSION  
UNIVERSITY-TYPE CONTRACTOR'S RECOMMENDATION FOR  
DISPOSITION OF SCIENTIFIC AND TECHNICAL DOCUMENT

( See Instructions on Reverse Side )

1. AEC REPORT NO.

C00-2118-0022

2. TITLE

Illiac III Computer System Manual:  
Taxicrinic Processor, Vol. 2

3. TYPE OF DOCUMENT (Check one):

- ☒ a. Scientific and technical report  
☐ b. Conference paper not to be published in a journal:

Title of conference \_\_\_\_\_

Date of conference \_\_\_\_\_

Exact location of conference \_\_\_\_\_

Sponsoring organization \_\_\_\_\_

- ☐ c. Other (Specify) \_\_\_\_\_

4. RECOMMENDED ANNOUNCEMENT AND DISTRIBUTION (Check one):

- ☒ a. AEC's normal announcement and distribution procedures may be followed.  
☐ b. Make available only within AEC and to AEC contractors and other U.S. Government agencies and their contractors.  
☐ c. Make no announcement or distribution.

REASON FOR RECOMMENDED RESTRICTIONS:

SUBMITTED BY: NAME AND POSITION (Please print or type)

Bernard J. Nordmann, Jr.  
Research Assistant

Organization

Department of Computer Science  
University of Illinois  
Urbana, Illinois 61801

Signature

*Bernard J. Nordmann Jr.*

Date

August 30, 1971

FOR AEC USE ONLY

AEC CONTRACT ADMINISTRATOR'S COMMENTS, IF ANY, ON ABOVE ANNOUNCEMENT AND DISTRIBUTION  
RECOMMENDATION:

8 PATENT CLEARANCE:

- ☐ a. AEC patent clearance has been granted by responsible AEC patent group.  
☐ b. Report has been sent to responsible AEC patent group for clearance.  
☐ c. Patent clearance not required.





OCT 13 1971















UNIVERSITY OF ILLINOIS-URBANA  
510.84 IL6R no. C002 no.475-480(1971  
Internal report /



3 0112 088400012